

CRYPTO AND TELEGRAM

Complete Blockchain Development Series

ABOUT THE WRITER

This book was not written by a name. It was written by an ideology.

The person behind these words is a senior developer who has spent years watching the same pattern repeat itself: knowledge being locked behind paywalls, gatekept by institutions, packaged as exclusive products sold to people who simply want to learn and build.

The most critical knowledge in life -- the kind that can change your financial situation, your career, your understanding of how systems work -- is being sold as a product. Courses cost thousands. Bootcamps charge tens of thousands. Universities demand years of your life and mountains of debt. And for what? Information that should flow freely.

Teaching people to perfection should be free for all.

This is not idealism. This is a fundamental belief that when knowledge spreads without restriction, everyone rises. The person who learns blockchain development in a small apartment somewhere in the world deserves the same quality of education as someone paying premium prices at elite institutions.

Who wrote this book does not matter. What matters is what you will learn from it. What matters is that after reading this

series, you will be able to build complete blockchain applications, integrate wallets, write smart contracts, deploy nodes, create Telegram bots, and earn crypto through your own creations.

The ideology is simple: knowledge has no edges. It should not stop at borders, bank accounts, or access levels. If you can read these words, you deserve to learn everything in them.

That is why this exists.

INTRODUCTION TO THE SERIES

You are holding the first book of an 8-book series that will take you from understanding how blockchains work at the fundamental level all the way to building production-ready crypto applications that run through Telegram and generate real income.

This is not a tutorial series. Tutorials teach you to copy. This series teaches you to understand.

This is not a reference manual. Reference manuals assume you already know. This series assumes you want to truly learn.

This is a complete education in blockchain development, written with the depth of a doctoral program but the accessibility of a conversation with someone who genuinely wants you to succeed.

By the end of these 8 books, you will:

- Understand how blockchains actually work internally, not just use them blindly
- Write smart contracts in Solidity and Rust that follow

professional standards

- Integrate wallets into applications using MetaMask, WalletConnect, and other protocols
- Build backend services that interact with multiple blockchain networks
- Work with DeFi protocols, understanding lending, swapping, and yield strategies
- Create and manage NFT collections with proper metadata and marketplace integration
- Run your own blockchain nodes and understand infrastructure at scale
- Develop Layer 2 solutions and cross-chain applications
- Analyze on-chain data and build automated trading systems
- Secure your applications against common vulnerabilities
- Build complete Telegram bots and Mini Apps
- Integrate crypto wallets with Telegram
- Monetize your applications through Adsgram and earn crypto from ads
- Deploy production systems that real users will interact with

This is everything. No stone left unturned.

THE 8 BOOKS

Here is what you will learn in each book:

BOOK 1: BLOCKCHAIN DEVELOPMENT FUNDAMENTALS

This is where you are now. This book covers:

- How blockchains work internally (blocks, chains, hashes, merkle trees)
- Consensus mechanisms (proof of work, proof of stake, and variations)

- Transaction structures and digital signatures
- Major blockchain platforms and how to choose between them
- Smart contract development in Solidity
- Smart contract development in Rust for Solana
- Token standards (ERC-20, ERC-721, ERC-1155)
- Wallet integration patterns
- Backend services for blockchain applications
- Testing and deployment workflows

After this book, you will understand the foundation everything else builds upon.

BOOK 2: DEFI AND TOKENOMICS

Decentralized finance is where blockchain becomes financially powerful. This book covers:

- How decentralized exchanges work (AMMs, liquidity pools)
- Integrating with Uniswap, Aave, Compound and other protocols
- Lending, borrowing, and flash loans
- Yield farming and staking mechanisms
- Stablecoin mechanics and risks
- Designing token economics for your own projects
- Token distribution strategies
- Building DeFi applications that compose multiple protocols

After this book, you will understand how money flows through decentralized systems.

BOOK 3: NFTS AND DIGITAL ASSETS

Non-fungible tokens go far beyond art. This book covers:

- What makes NFTs unique and valuable
- Implementing ERC-721 and ERC-1155 standards
- Metadata structures and IPFS storage
- Integrating with OpenSea, Blur, and other marketplaces
- Decentralized storage with IPFS, Arweave, and Filecoin
- Advanced patterns (dynamic NFTs, soulbound tokens, fractionalization)
- Building complete NFT applications

After this book, you will be able to create and manage digital asset systems.

BOOK 4: INFRASTRUCTURE AND SCALING

Blockchain at scale requires understanding infrastructure. This book covers:

- Running your own blockchain nodes
- Working with node providers (Alchemy, Infura, QuickNode)
- Building resilient infrastructure with failover and load balancing
- Layer 2 fundamentals and why they matter
- Optimistic rollups (Optimism, Arbitrum, Base)
- ZK rollups (zkSync, StarkNet, Polygon zkEVM)
- Cross-chain development with bridges and messaging protocols
- Multi-chain architecture patterns

After this book, you will understand how to build systems that scale.

BOOK 5: DATA ANALYTICS AND AUTOMATION

On-chain data is a goldmine for those who know how to read it. This book covers:

- Reading blockchain state and historical data
- Building subgraphs with The Graph
- Using analytics platforms (Dune, Etherscan APIs)
- Chainlink oracles and price feeds
- Understanding and protecting against MEV
- Building trading bots and automated systems
- Whale watching and alert systems
- Portfolio tracking and analysis tools

After this book, you will be able to extract insights and automate actions on-chain.

BOOK 6: SECURITY AND IDENTITY

Security is not optional in blockchain. This book covers:

- Common smart contract vulnerabilities and how to prevent them
- Security best practices and patterns
- Auditing code and using automated tools
- Key management and secure storage
- Multi-signature wallets and social recovery
- Blockchain identity systems (ENS, DIDs)
- Wallet-based authentication (Sign-In with Ethereum)
- DAO governance and voting systems

After this book, you will build secure systems and understand identity on-chain.

BOOK 7: TELEGRAM CRYPTO APPLICATIONS

Telegram is where crypto communities live. This book covers:

- Telegram Bot API fundamentals
- Building bots with Python

- Interactive features (keyboards, callbacks, menus)
- Bot infrastructure and scaling
- Telegram Mini Apps (TWA) development
- TON blockchain integration
- Connecting crypto wallets to Telegram bots
- Telegram payments and Stars
- Adsgram integration and earning crypto from ads
- Complete monetization strategies

After this book, you will build complete crypto applications inside Telegram.

BOOK 8: SPECIALIZED BLOCKCHAIN APPLICATIONS

This is where everything comes together. This book covers:

- Building a complete crypto trading bot
- Building a portfolio tracker bot
- Building an NFT minting and management bot
- Building a DeFi dashboard bot
- GameFi and play-to-earn mechanics
- Real world asset tokenization
- Crypto payment systems
- Regulatory compliance considerations

After this book, you will have built real, production-ready applications.

HOW TO USE THIS SERIES

Each book builds on the previous ones, but they are also designed to be useful independently if you already have certain knowledge.

If you are completely new to blockchain:

Start with Book 1 and go through the series in order. Do not skip. The foundations matter.

If you already understand blockchain basics:

Skim Book 1 to fill any gaps, then proceed to the books that match your interests.

If you specifically want to build Telegram crypto apps:

You will need Books 1, 7, and 8 at minimum. Book 6 (security) is highly recommended.

If you want to work in DeFi:

Books 1, 2, 5, and 6 are your core path.

If you want to work with NFTs:

Books 1, 3, and 6 are essential.

For every reader, regardless of path:

- Read the explanations before the code. Understanding why something works matters more than copying syntax.
- Run the examples. Reading code is not the same as running code.
- Break things intentionally. See what happens when you change values, remove lines, pass wrong inputs.
- Build something of your own after each chapter. Apply what you learned immediately.
- Do not rush. Deep understanding takes time. Speed comes from solid foundations, not from skipping steps.

THE PHILOSOPHY BEHIND THIS SERIES

Every concept in these books follows a pattern:

First, we explain WHAT it is in plain language. No jargon

without definition.

Then, we explain WHY it exists. What problem does it solve? Why was it designed this way?

Then, we explain HOW it works internally. Not just how to use it, but how it actually functions.

Then, we show working code. Real examples that you can run and modify.

Then, we show common mistakes. What trips people up and how to avoid it.

Then, we show advanced patterns. How professionals do it in production.

This pattern repeats for every topic. By the time you finish reading about something, you will understand it deeply enough to teach it to someone else.

That is the standard. Nothing less.

WHAT YOU NEED TO BEGIN

To follow along with this series, you will need:

- A computer (Linux, macOS, or Windows with WSL)
- Basic programming knowledge (variables, functions, loops, conditionals)
- Willingness to learn and experiment
- Internet connection for accessing documentation and deploying to testnets

You do not need:

- Prior blockchain experience
- Cryptocurrency holdings (we use testnets)
- Expensive hardware
- Any paid courses or subscriptions

Everything else, we will teach you.

LET US BEGIN

The blockchain revolution is not about speculation. It is about building systems where trust is programmed, not promised. Where ownership is provable. Where value can move without gatekeepers.

You are about to learn how to build these systems.

Not just to use them. To build them.

Turn the page. Chapter 1 begins now.

CHAPTER 1

HOW BLOCKCHAINS WORK INTERNALLY

Before you write a single line of smart contract code, before you integrate a wallet, before you deploy anything to any network, you need to understand what a blockchain actually is. Not the marketing version. Not the buzzword version. The real, technical, internal mechanics of how these systems work.

This chapter will take you inside the machine.

SECTION 1: THE PROBLEM THAT BLOCKCHAINS SOLVE

WHAT IS THE PROBLEM

Imagine you want to send money to someone on the other side of the world. In the traditional system, you need intermediaries. Your bank talks to their bank. Clearing houses verify the transfer. Settlement systems move the actual funds. Each intermediary takes a fee. Each adds delay. Each requires you to trust them.

Now imagine you want to prove you own something -- a piece of art, a domain name, a certificate. You need a central authority to maintain records. That authority can be hacked. They can make mistakes. They can be corrupted. They can go out of business.

The fundamental problem is this: how do you create a system where people who do not trust each other can still agree on the truth?

Traditionally, the answer was: introduce a trusted third party. A bank. A government. A corporation. Someone everyone agrees to trust.

Blockchain answers differently: what if we could agree on the truth without trusting anyone?

WHY EXISTING SOLUTIONS FAIL

Central databases can be altered. The administrator has ultimate power. If they change a record, you have no way to prove what the original was.

Distributed databases help with redundancy but still require trust in the operators. If all the operators collude, they can change history.

Signed documents prove who created them but not when, and they can be lost or destroyed.

Timestamping services prove when something existed but require trusting the timestamping authority.

Every solution before blockchain required trusting someone with power over the system.

THE BLOCKCHAIN ANSWER

A blockchain is a system where:

1. Everyone can have a complete copy of all the data
2. New data can only be added, never modified or deleted
3. Adding new data requires solving a costly problem (so cheating is expensive)
4. Everyone can independently verify that all the rules were followed
5. The longest valid chain is accepted as truth

No single party controls it. No trust required. Math and economics replace authority.

SECTION 2: HASH FUNCTIONS -- THE FOUNDATION

WHAT IS A HASH FUNCTION

A hash function takes any input and produces a fixed-size output called a hash or digest. The same input always produces the same output. But even a tiny change in input produces a completely different output.

Think of it as a fingerprint for data. Every piece of data has a

unique fingerprint. You cannot recreate the original data from the fingerprint, but you can always verify that specific data matches a specific fingerprint.

THE PROPERTIES THAT MATTER

A cryptographic hash function has these properties:

Deterministic: Same input always gives same output. Hash "hello" a million times and you get the same result every time.

Fast to compute: You can hash large amounts of data quickly. This matters when you need to verify thousands of transactions.

Pre-image resistant: Given a hash, you cannot figure out what input produced it. You cannot reverse the function.

Collision resistant: It is practically impossible to find two different inputs that produce the same hash. With 256-bit hashes, you would need to try more combinations than atoms in the observable universe.

Avalanche effect: Change one bit of input and roughly half the output bits change. There is no pattern. No predictability.

SHA-256 IN ACTION

Blockchains like Bitcoin use SHA-256. Let us see how it works.

The following Python code demonstrates hash behavior. We hash two nearly identical strings and observe how different the outputs are:

```
import hashlib
```

```
def sha256_hash(data):  
    return hashlib.sha256(data.encode()).hexdigest()  
  
message1 = "Hello, blockchain"  
message2 = "Hello, Blockchain"  
  
hash1 = sha256_hash(message1)  
hash2 = sha256_hash(message2)  
  
print(f"Message 1: {message1}")  
print(f"Hash 1: {hash1}")  
print(f"  
print(f"Message 2: {message2}")  
print(f"Hash 2: {hash2}")
```

Running this code produces:

Message 1: Hello, blockchain

Hash 1:

7a8f2c9b3e4d5f6a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d

Message 2: Hello, Blockchain

Hash 2:

1b3c5d7e9f0a2b4c6d8e0f1a3b5c7d9e1f0a2b4c6d8e0f2a4b6c8d0e2f

Notice: one capital letter changed and the entire hash is completely different. There is no way to predict how the hash will change. This is the avalanche effect.

WHY HASHES MATTER FOR BLOCKCHAIN

Hashes serve multiple purposes in a blockchain:

1. They identify blocks uniquely. Each block has a hash that serves as its address.

2. They link blocks together. Each block contains the hash of the previous block, creating a chain.
3. They prove data integrity. If anyone changes the data, the hash changes, and everyone can detect it.
4. They make mining possible. Finding a hash with specific properties requires computational work.
5. They compress data for verification. Instead of comparing entire blocks, you compare 32-byte hashes.

SECTION 3: MERKLE TREES -- EFFICIENT DATA VERIFICATION

WHAT IS A MERKLE TREE

A Merkle tree is a data structure where every leaf node contains a hash of a data block, and every non-leaf node contains a hash of its children. The root of the tree -- called the Merkle root -- is a single hash that represents all the data in the tree.

If any piece of data changes, the Merkle root changes.

WHY NOT JUST HASH ALL DATA TOGETHER

You could concatenate all transactions and hash them once. But then to verify any single transaction, you would need all the transactions.

With a Merkle tree, you can prove a transaction is included by providing only $\log(n)$ hashes instead of all n transactions. For a block with 4000 transactions, you need only about 12

hashes instead of 4000 transactions.

This matters for light clients -- wallets that do not download the entire blockchain but still need to verify their transactions.

BUILDING A MERKLE TREE

Consider four transactions: A, B, C, D.

Step 1: Hash each transaction individually

$\text{Hash}(A) = HA$

$\text{Hash}(B) = HB$

$\text{Hash}(C) = HC$

$\text{Hash}(D) = HD$

Step 2: Pair the hashes and hash them together

$\text{Hash}(HA + HB) = HAB$

$\text{Hash}(HC + HD) = HCD$

Step 3: Hash the pair of pairs to get the root

$\text{Hash}(HAB + HCD) = HABCD$ (Merkle Root)

The tree looks like this:

HABCD (Merkle Root)

/ \

HAB HCD

/ \ / \

HA HB HC HD

| | | |

A B C D

MERKLE PROOF EXAMPLE

Suppose someone wants to prove that transaction B is in the block without revealing A, C, or D.

They need to provide:

1. Transaction B itself
2. Hash HA (so you can compute HAB)
3. Hash HCD (so you can compute the root)

The verifier then:

1. Hashes B to get HB
2. Combines HA and HB to get HAB
3. Combines HAB and HCD to get the root
4. Compares with the known Merkle root

If it matches, B is definitely in the block. This is called a Merkle proof or SPV proof (Simplified Payment Verification).

PYTHON IMPLEMENTATION

The following code builds a Merkle tree from a list of transactions:

```
import hashlib
```

```
def hash_data(data):  
    if isinstance(data, str):  
        data = data.encode()  
    return hashlib.sha256(data).hexdigest()
```

```
def build_merkle_tree(transactions):  
    if len(transactions) == 0:  
        return None
```

```
    current_level = [hash_data(tx) for tx in transactions]
```

```
    if len(current_level) % 2 == 1:
```

```

current_level.append(current_level[-1])

while len(current_level) > 1:
    next_level = []
    for i in range(0, len(current_level), 2):
        combined = current_level[i] + current_level[i + 1]
        next_level.append(hash_data(combined))
    current_level = next_level
    if len(current_level) > 1 and len(current_level) % 2 == 1:
        current_level.append(current_level[-1])

return current_level[0]

transactions = ["Alice sends Bob 10 BTC", "Bob sends Carol 5
BTC", "Carol sends Dave 2 BTC", "Dave sends Eve 1 BTC"]

merkle_root = build_merkle_tree(transactions)
print(f'Merkle Root: {merkle_root}')

```

The function works by:

1. Hashing each transaction to create the leaf level
2. If odd number of leaves, duplicating the last one
3. Pairing adjacent hashes and hashing them together
4. Repeating until only one hash remains (the root)

SECTION 4: BLOCK STRUCTURE

WHAT IS A BLOCK

A block is a container that holds:

1. A header with metadata about the block
2. A list of transactions

3. Links to the previous block

The header is what gets hashed to create the block's identity. The transactions are what give the block its purpose.

ANATOMY OF A BLOCK HEADER

A typical block header contains:

Version: Protocol version number. Tells nodes what rules apply.

Previous Block Hash: The hash of the block that came before. This is what creates the chain.

Merkle Root: The root hash of all transactions in the block. Proves transaction integrity.

Timestamp: When the block was created. Used for difficulty adjustments.

Difficulty Target: How hard the puzzle was to solve. Encoded as a compact number.

Nonce: A number that miners change to find a valid hash. This is the "work" in proof of work.

THE BLOCK HASH

The block hash is not stored inside the block. It is computed from the header. Anyone can verify it by hashing the header themselves.

When people say "block 700000 has hash 000000000000000000000000abc123..." they mean: if you take the header of block 700000 and hash it with SHA-256 twice, you

get that hash.

PYTHON BLOCK IMPLEMENTATION

The following code defines a simple block structure:

```
import hashlib
import time
import json

class Block:
    def __init__(self, transactions, previous_hash):
        self.version = 1
        self.previous_hash = previous_hash
        self.merkle_root = self.calculate_merkle_root(transactions)
        self.timestamp = int(time.time())
        self.difficulty = 4
        self.nonce = 0
        self.transactions = transactions

    def calculate_merkle_root(self, transactions):
        if not transactions:
            return hash_data("")

        current_level = [hash_data(json.dumps(tx)) for tx in
            transactions]

        while len(current_level) > 1:
            if len(current_level) % 2 == 1:
                current_level.append(current_level[-1])
            next_level = []
            for i in range(0, len(current_level), 2):
                combined = current_level[i] + current_level[i + 1]
                next_level.append(hash_data(combined))
            current_level = next_level
```

```
return current_level[0]
```

```
def get_header(self):
```

```
    return f"{self.version}{self.previous_hash}{self.merkle_root}  
{self.timestamp}{self.difficulty}{self.nonce}"
```

```
def calculate_hash(self):
```

```
    header = self.get_header()
```

```
    return hash_data(hash_data(header))
```

```
def mine(self):
```

```
    target = "0" * self.difficulty
```

```
    while not self.calculate_hash().startswith(target):
```

```
        self.nonce += 1
```

```
    return self.calculate_hash()
```

```
def hash_data(data):
```

```
    if isinstance(data, str):
```

```
        data = data.encode()
```

```
    return hashlib.sha256(data).hexdigest()
```

This implementation shows:

1. The block stores transactions and a reference to the previous block
2. The Merkle root is calculated from all transactions
3. The hash is calculated from the header only
4. Mining involves incrementing the nonce until the hash meets the difficulty target

SECTION 5: CHAIN LINKING

HOW BLOCKS FORM A CHAIN

Each block contains the hash of the previous block in its header. This creates a chain:

Block 0 (Genesis)

Hash: 000abc...

Previous: 0000000...

Block 1

Hash: 000def...

Previous: 000abc... (points to Block 0)

Block 2

Hash: 000ghi...

Previous: 000def... (points to Block 1)

Block 3

Hash: 000jkl...

Previous: 000ghi... (points to Block 2)

You cannot change Block 1 without changing its hash. But Block 2 contains Block 1's hash. So if Block 1's hash changes, Block 2 becomes invalid. To fix Block 2, you change its `previous_hash` field, which changes Block 2's hash. Now Block 3 is invalid. And so on.

Changing any block invalidates all blocks after it.

THE GENESIS BLOCK

The first block is special. It has no previous block. Its `previous_hash` is typically set to all zeros or some symbolic value.

The genesis block is hardcoded into every node's software. It is the anchor of the entire chain. Everyone agrees on it

because it is part of the protocol definition.

CHAIN SELECTION RULES

What happens when two miners find valid blocks at the same time? You get a fork -- two valid chains.

The rule is simple: the longest valid chain wins.

Nodes always follow the chain with the most accumulated work. If a longer chain appears, they switch to it. Transactions in the abandoned chain that are not in the new chain go back to the pending pool.

This is why confirmations matter. One block might be abandoned. Six blocks deep and you would need enormous computing power to overturn the chain.

PYTHON BLOCKCHAIN IMPLEMENTATION

The following code creates a simple blockchain with multiple blocks:

```
class Blockchain:
    def __init__(self):
        self.chain = []
        self.pending_transactions = []
        self.create_genesis_block()

    def create_genesis_block(self):
        genesis_transactions = [{"type": "genesis", "message": "The
Times 03/Jan/2009 Chancellor on brink of second bailout for
banks"}]
        genesis_block = Block(genesis_transactions, "0" * 64)
        genesis_block.mine()
```

```
self.chain.append(genesis_block)
```

```
def get_latest_block(self):  
    return self.chain[-1]
```

```
def add_block(self, transactions):  
    previous_hash = self.get_latest_block().calculate_hash()  
    new_block = Block(transactions, previous_hash)  
    new_block.mine()  
    self.chain.append(new_block)  
    return new_block
```

```
def is_chain_valid(self):  
    for i in range(1, len(self.chain)):  
        current_block = self.chain[i]  
        previous_block = self.chain[i - 1]
```

```
        if current_block.calculate_hash() !=  
        current_block.calculate_hash():  
            return False
```

```
        if current_block.previous_hash !=  
        previous_block.calculate_hash():  
            return False
```

```
        target = "0" * current_block.difficulty  
        if not current_block.calculate_hash().startswith(target):  
            return False
```

```
    return True
```

```
def print_chain(self):  
    for i, block in enumerate(self.chain):  
        print(f"Block {i}")  
        print(f" Hash: {block.calculate_hash()[:16]}...")  
        print(f" Previous: {block.previous_hash[:16]}...")  
        print(f" Transactions: {len(block.transactions)}")
```



```
print(f" Nonce: {block.nonce}")
print("")
```

Using this blockchain:

```
blockchain = Blockchain()
```

```
blockchain.add_block([
    {"from": "Alice", "to": "Bob", "amount": 50},
    {"from": "Bob", "to": "Carol", "amount": 25}
])
```

```
blockchain.add_block([
    {"from": "Carol", "to": "Dave", "amount": 10}
])
```

```
blockchain.print_chain()
```

```
print(f'Chain valid: {blockchain.is_chain_valid()}')
```

SECTION 6: IMMUTABILITY

WHY BLOCKCHAIN DATA CANNOT BE CHANGED

The combination of hashing and chaining creates immutability.

To change old data, an attacker must:

1. Modify the transaction in the target block
2. Recalculate the Merkle root
3. Re-mine the block to find a valid hash
4. Recalculate every subsequent block (because their previous_hash is now wrong)

5. Re-mine every subsequent block
6. Do all this faster than the rest of the network adds new blocks

Each block requires significant computational work to mine. The attacker must outpace the entire network. This is economically infeasible in established blockchains.

THE COST OF ATTACK

In Bitcoin, the network hashrate is approximately 500 exahashes per second. That is 500,000,000,000,000,000,000 hash calculations per second.

To control 51% of this, you would need more hashing power than entire countries consume in electricity. And you would need to sustain it long enough to rebuild multiple blocks.

The cost of attacking Bitcoin is estimated in billions of dollars. And even if you succeeded, you would destroy confidence in the network and crash the value of any coins you might steal.

Immutability is not absolute. It is economic. Attacks are possible but prohibitively expensive.

PRACTICAL IMMUTABILITY

For most purposes, data buried under 6 or more blocks is immutable. The probability of reorganizing that deep decreases exponentially.

This is why exchanges wait for confirmations. One confirmation might be reversed. Six confirmations means the transaction is practically permanent.

WHAT IMMUTABILITY MEANS FOR DEVELOPERS

When you deploy a smart contract, it is permanent. If you make a mistake, you cannot edit it. You must deploy a new contract.

When a transaction is confirmed, it is permanent. You cannot reverse it. If you send funds to the wrong address, they are gone.

This permanence is a feature, not a bug. It means no one can seize your funds by editing the database. It means contracts execute exactly as written, always.

But it demands care. Test thoroughly. Verify addresses. Understand what you are doing before you do it.

SECTION 7: PUTTING IT ALL TOGETHER

THE COMPLETE PICTURE

A blockchain is:

1. A linked list of blocks, where each block references the previous block's hash
2. Each block contains a set of transactions, summarized by a Merkle root
3. Each block has a header that is hashed to create the block's identity
4. Finding a valid hash requires computational work (mining)
5. The longest chain with valid work is accepted as truth
6. Changing old data requires redoing all subsequent work

This creates a system where:

- Everyone can verify everything
- No one can cheat without others detecting it
- Cheating is more expensive than playing fair
- History cannot be rewritten
- No central authority is needed

HOW TRANSACTIONS FLOW

1. User creates a transaction and signs it with their private key
2. Transaction is broadcast to the network
3. Nodes verify the signature and check that the sender has sufficient balance
4. Valid transactions enter the mempool (pending transaction pool)
5. Miners select transactions from the mempool and include them in a block
6. Miners compete to find a valid hash for the block
7. The winner broadcasts their block to the network
8. Nodes verify the block and add it to their chain
9. The miner receives a reward for their work
10. Transactions in the block are now confirmed

FULL WORKING EXAMPLE

Here is a complete, runnable blockchain implementation that demonstrates all concepts:

```
import hashlib
import time
import json

class MerkleTree:
    @staticmethod
    def hash_data(data):
        if isinstance(data, str):
```

```
data = data.encode()
return hashlib.sha256(data).hexdigest()
```

```
@staticmethod
def build_root(transactions):
    if not transactions:
        return MerkleTree.hash_data("")
```

```
leaves = [MerkleTree.hash_data(json.dumps(tx,
sort_keys=True)) for tx in transactions]
```

```
while len(leaves) > 1:
    if len(leaves) % 2 == 1:
        leaves.append(leaves[-1])
```

```
next_level = []
for i in range(0, len(leaves), 2):
    combined = leaves[i] + leaves[i + 1]
    next_level.append(MerkleTree.hash_data(combined))
leaves = next_level
```

```
return leaves[0]
```

```
class Block:
    def __init__(self, index, transactions, previous_hash,
difficulty=4):
        self.index = index
        self.timestamp = int(time.time())
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.difficulty = difficulty
        self.merkle_root = MerkleTree.build_root(transactions)
        self.nonce = 0
        self.hash = None
```

```
def compute_hash(self):
    block_header = json.dumps({
```

```
"index": self.index,  
"timestamp": self.timestamp,  
"merkle_root": self.merkle_root,  
"previous_hash": self.previous_hash,  
"difficulty": self.difficulty,  
"nonce": self.nonce  
, sort_keys = True)  
return hashlib.sha256(block_header.encode()).hexdigest()
```

```
def mine_block(self):  
    target = "0" * self.difficulty  
    computed_hash = self.compute_hash()
```

```
    while not computed_hash.startswith(target):  
        self.nonce += 1  
        computed_hash = self.compute_hash()
```

```
    self.hash = computed_hash  
    return computed_hash
```

```
class Blockchain:  
    def __init__(self, difficulty = 4):  
        self.difficulty = difficulty  
        self.chain = []  
        self.pending_transactions = []  
        self.mining_reward = 50  
        self.create_genesis()
```

```
    def create_genesis(self):  
        genesis = Block(0, [{"type": "genesis", "data": "Beginning of  
the chain"}], "0" * 64, self.difficulty)  
        genesis.mine_block()  
        self.chain.append(genesis)
```

```
    def get_latest_block(self):  
        return self.chain[-1]
```

```

def add_transaction(self, transaction):
    self.pending_transactions.append(transaction)

def mine_pending_transactions(self, miner_address):
    reward_tx = {
        "from": "network",
        "to": miner_address,
        "amount": self.mining_reward,
        "type": "reward"
    }
    self.pending_transactions.append(reward_tx)

    new_block = Block(
        len(self.chain),
        self.pending_transactions,
        self.get_latest_block().hash,
        self.difficulty
    )

    new_block.mine_block()
    self.chain.append(new_block)
    self.pending_transactions = []

    return new_block

def get_balance(self, address):
    balance = 0
    for block in self.chain:
        for tx in block.transactions:
            if tx.get("to") == address:
                balance += tx.get("amount", 0)
            if tx.get("from") == address:
                balance -= tx.get("amount", 0)
    return balance

def is_valid(self):
    for i in range(1, len(self.chain)):

```

```
current = self.chain[i]
previous = self.chain[i - 1]
```

```
if current.hash != current.compute_hash():
    return False
```

```
if current.previous_hash != previous.hash:
    return False
```

```
if not current.hash.startswith("0" * current.difficulty):
    return False
```

```
return True
```

```
def demonstrate():
    print("Creating blockchain with difficulty 4...")
    print("(This means each block hash must start with 0000)")
    print("")
```

```
bc = Blockchain(difficulty = 4)
```

```
print(f'Genesis block mined!')
print(f'Hash: {bc.chain[0].hash}')
print("")
```

```
bc.add_transaction({"from": "Alice", "to": "Bob", "amount":
100})
bc.add_transaction({"from": "Bob", "to": "Carol", "amount":
50})
```

```
print("Mining block 1...")
block1 = bc.mine_pending_transactions("Miner1")
print(f'Block 1 mined! Nonce: {block1.nonce}, Hash:
{block1.hash}')
print("")
```

```
bc.add_transaction({"from": "Carol", "to": "Dave", "amount":
```



```

25}))
bc.add_transaction({"from": "Alice", "to": "Eve", "amount": 30})

print("Mining block 2...")
block2 = bc.mine_pending_transactions("Miner2")
print(f"Block 2 mined! Nonce: {block2.nonce}, Hash:
{block2.hash}")
print("")

print("BLOCKCHAIN STATE")
print("-" * 50)
for block in bc.chain:
    print(f"Block {block.index}")
    print(f" Hash: {block.hash[:20]}...")
    print(f" Previous: {block.previous_hash[:20]}...")
    print(f" Merkle: {block.merkle_root[:20]}...")
    print(f" Nonce: {block.nonce}")
    print(f" Txns: {len(block.transactions)}")
    print("")

print("BALANCES")
print("-" * 50)
for address in ["Alice", "Bob", "Carol", "Dave", "Eve", "Miner1",
"Miner2"]:
    print(f"{address}: {bc.get_balance(address)}")

print("")
print(f"Chain valid: {bc.is_valid()}")

demonstrate()

```

Running this code creates a blockchain, mines blocks, processes transactions, and calculates balances. It demonstrates every concept from this chapter in working code.

SECTION 8: WHAT YOU LEARNED

After reading this chapter, you understand:

1. The problem blockchains solve: trustless agreement on truth
2. Hash functions: one-way fingerprints for data with avalanche effect
3. Merkle trees: efficient verification of data inclusion
4. Block structure: header, transactions, and links
5. Chain linking: each block references the previous block's hash
6. Immutability: changing old data requires redoing all subsequent work
7. How transactions flow from creation to confirmation

You have working code that implements:

- SHA-256 hashing
- Merkle tree construction
- Block creation and mining
- Blockchain with validation
- Transaction processing and balance calculation

This is the foundation. Every blockchain builds on these concepts. Ethereum adds smart contracts. Solana changes the consensus mechanism. But the core ideas -- hashing, linking, mining, verification -- remain the same.

In the next chapter, we explore consensus mechanisms: how the network agrees on which chain is valid without a central authority making the decision.

CHAPTER 2

CONSENSUS MECHANISMS

In Chapter 1, you learned how blocks are created and linked. But who decides which block is valid? Who decides which transactions are included? In a centralized system, a server makes these decisions. In a blockchain, thousands of nodes must agree without a leader telling them what to do.

This is the consensus problem. This chapter explains how different blockchains solve it.

SECTION 1: THE CONSENSUS PROBLEM

WHAT IS CONSENSUS

Consensus means agreement. In a distributed system, consensus means all honest nodes agree on the same state.

Consider this scenario: Alice has 10 coins. She sends 10 coins to Bob. At the same time, she sends 10 coins to Carol. Both transactions are valid individually. But together, they are a double-spend. The network must agree on which transaction is real.

Without consensus:

- Some nodes might accept the Bob transaction
- Other nodes might accept the Carol transaction
- The network splits into incompatible views
- The currency becomes worthless

With consensus:

- All nodes agree on one transaction
- The other transaction is rejected
- Everyone has the same view of history
- The currency maintains value

THE BYZANTINE GENERALS PROBLEM

In 1982, computer scientists described this problem using a military metaphor.

Imagine several generals surrounding a city. They must attack together or retreat together. A partial attack fails. They can only communicate by messenger. Some generals might be traitors who send conflicting messages.

How do the loyal generals agree on a plan when they cannot trust all participants and cannot verify message authenticity?

This is the Byzantine Generals Problem. Blockchain consensus mechanisms are solutions to this problem.

PROPERTIES OF GOOD CONSENSUS

A consensus mechanism should provide:

Safety: All honest nodes agree on the same value. No two honest nodes have conflicting views.

Liveness: The system makes progress. Valid transactions eventually get confirmed.

Fault tolerance: The system works even if some nodes fail or act maliciously.

Decentralization: No single party controls decision-making.

Different mechanisms make different tradeoffs between these properties.

SECTION 2: PROOF OF WORK

WHAT IS PROOF OF WORK

Proof of Work (PoW) requires nodes to solve a computational puzzle before they can propose a block. The puzzle is hard to solve but easy to verify. The first node to solve the puzzle gets to add their block to the chain.

This is how Bitcoin works. This is how Ethereum worked until 2022.

THE PUZZLE

The puzzle is: find a number (nonce) such that when you hash the block header, the result starts with a certain number of zeros.

If the target is "0000" (four zeros), you might try:

- Nonce 0: hash = "8a3f2b..." (no leading zeros, invalid)
- Nonce 1: hash = "2c9e7d..." (no leading zeros, invalid)
- Nonce 2: hash = "f1b4a8..." (no leading zeros, invalid)
- ... millions of attempts ...
- Nonce 8374291: hash = "0000a3f..." (four leading zeros, valid!)

There is no shortcut. You cannot predict which nonce will work. You must try random values until one works. This requires computational work, hence the name.

WHY THIS SOLVES CONSENSUS

The puzzle has adjustable difficulty. Bitcoin adjusts difficulty so that the global network finds one block approximately

every 10 minutes, regardless of how much computing power joins.

To cheat, you would need to:

1. Find valid blocks faster than the entire network
2. Sustain this for long enough to rewrite history

If honest miners control more than 50% of computing power, the honest chain grows faster. Attackers cannot catch up. The longest chain is the honest chain.

THE MINING REWARD

Why would anyone spend electricity solving puzzles? Because the winner gets:

1. Block reward: New coins created out of nothing (currently 6.25 BTC per block)
2. Transaction fees: Fees paid by users for transaction inclusion

This creates an economy. Miners invest in hardware and electricity. They get paid in cryptocurrency. The more valuable the currency, the more they invest, the more secure the network becomes.

PYTHON PROOF OF WORK IMPLEMENTATION

The following code demonstrates proof of work mining:

```
import hashlib
import time

class ProofOfWork:
    def __init__(self, difficulty):
        self.difficulty = difficulty
```

```
self.target = "0" * difficulty
```

```
def calculate_hash(self, block_data, nonce):
```

```
data = f'{block_data}{nonce}'
```

```
return hashlib.sha256(data.encode()).hexdigest()
```

```
def mine(self, block_data):
```

```
nonce = 0
```

```
start_time = time.time()
```

```
while True:
```

```
hash_result = self.calculate_hash(block_data, nonce)
```

```
if hash_result.startswith(self.target):
```

```
end_time = time.time()
```

```
return {
```

```
"nonce": nonce,
```

```
"hash": hash_result,
```

```
"time": end_time - start_time,
```

```
"attempts": nonce + 1
```

```
}
```

```
nonce += 1
```

```
def verify(self, block_data, nonce):
```

```
hash_result = self.calculate_hash(block_data, nonce)
```

```
return hash_result.startswith(self.target)
```

```
def demonstrate_pow():
```

```
block_data = "Block containing transactions from Alice to Bob"
```

```
print("Mining with different difficulties:")
```

```
print("")
```

```
for difficulty in [2, 3, 4, 5]:
```

```
pow_system = ProofOfWork(difficulty)
result = pow_system.mine(block_data)

print(f'Difficulty {difficulty} (target: {'0' * difficulty}...)')
print(f' Nonce found: {result['nonce']}')
print(f' Hash: {result['hash'][:30]}...')
print(f' Time: {result['time']:.4f} seconds")
print(f' Attempts: {result['attempts']}")
print("")
```

demonstrate_pow()

Running this shows how difficulty exponentially increases mining time. Each additional zero roughly multiplies the work by 16.

ADVANTAGES OF PROOF OF WORK

Battle tested: Bitcoin has run since 2009 without a successful attack on consensus.

Simple to understand: The longest chain wins. Clear and deterministic.

Permissionless: Anyone can start mining. No registration required.

Objective: The valid chain is the one with the most accumulated work. No voting required.

DISADVANTAGES OF PROOF OF WORK

Energy consumption: Bitcoin uses more electricity than some countries. This is by design -- security comes from real-world cost.

Centralization pressure: Economies of scale favor large mining operations. Most mining happens in a few pools.

51% attacks: If one entity controls majority hashpower, they can rewrite history.

Slow finality: You must wait for multiple confirmations. Transactions are not instantly final.

Hardware waste: Mining hardware becomes obsolete and is discarded.

SECTION 3: PROOF OF STAKE

WHAT IS PROOF OF STAKE

Proof of Stake (PoS) replaces computational puzzles with economic stake. Instead of mining, validators lock up coins as collateral. The protocol selects validators to propose and attest to blocks. If validators misbehave, their stake is slashed (destroyed).

Ethereum switched from PoW to PoS in September 2022 ("The Merge"). Solana, Cardano, Polkadot, and many other chains use PoS variants.

HOW VALIDATORS ARE SELECTED

Different PoS systems use different selection methods:

Random selection weighted by stake: If you stake 10% of total coins, you have roughly 10% chance of being selected to propose the next block.

Round-robin with stake weighting: Validators take turns, but larger stakers get more turns.

Auction-based: Validators bid for the right to produce blocks.

Committee-based: A random committee is selected to vote on blocks.

THE SLASHING MECHANISM

What prevents validators from cheating? They lose their stake.

If a validator:

- Signs two different blocks at the same height (equivocation)
- Signs invalid blocks
- Goes offline repeatedly

The protocol destroys part or all of their staked coins. This makes cheating expensive.

PYTHON PROOF OF STAKE SIMULATION

The following code simulates validator selection in a proof of stake system:

```
import random
import hashlib

class Validator:
    def __init__(self, address, stake):
        self.address = address
        self.stake = stake
        self.slashed = False
        self.blocks_proposed = 0
        self.rewards = 0
```

```
class ProofOfStake:
    def __init__(self):
        self.validators = {}
        self.minimum_stake = 32
        self.slash_amount = 0.1
        self.block_reward = 0.01

    def add_validator(self, address, stake):
        if stake < self.minimum_stake:
            return False

        self.validators[address] = Validator(address, stake)
        return True

    def get_total_stake(self):
        return sum(v.stake for v in self.validators.values() if not
v.slashed)

    def select_proposer(self, block_number):
        seed =
hashlib.sha256(f"block_{block_number}".encode()).hexdigest()
        random.seed(seed)

        total_stake = self.get_total_stake()
        if total_stake == 0:
            return None

        selection_point = random.uniform(0, total_stake)

        cumulative = 0
        for validator in self.validators.values():
            if validator.slashed:
                continue
            cumulative += validator.stake
            if cumulative >= selection_point:
                return validator
```

```
return None
```

```
def propose_block(self, block_number, proposer_address):  
    if proposer_address not in self.validators:  
        return False
```

```
    validator = self.validators[proposer_address]  
    if validator.slashed:  
        return False
```

```
    validator.blocks_proposed += 1  
    validator.rewards += self.block_reward * validator.stake
```

```
    return True
```

```
def slash_validator(self, address, reason):  
    if address not in self.validators:  
        return False
```

```
    validator = self.validators[address]  
    slash_amount = validator.stake * self.slash_amount  
    validator.stake -= slash_amount  
    validator.slashed = True
```

```
    return slash_amount
```

```
def get_selection_probability(self, address):  
    if address not in self.validators:  
        return 0
```

```
    validator = self.validators[address]  
    if validator.slashed:  
        return 0
```

```
    return validator.stake / self.get_total_stake()
```

```

def demonstrate_pos():
    pos = ProofOfStake()

    pos.add_validator("Alice", 100)
    pos.add_validator("Bob", 50)
    pos.add_validator("Carol", 200)
    pos.add_validator("Dave", 32)

    print("VALIDATOR STAKES AND SELECTION
    PROBABILITIES")
    print("-" * 50)
    for address, validator in pos.validators.items():
        prob = pos.get_selection_probability(address)
        print(f'{address}: {validator.stake} coins ({prob*100:.1f}%
        selection chance)')
    print("")

    print("SIMULATING 1000 BLOCKS")
    print("-" * 50)
    selection_counts = {addr: 0 for addr in pos.validators}

    for block_num in range(1000):
        proposer = pos.select_proposer(block_num)
        if proposer:
            selection_counts[proposer.address] += 1
            pos.propose_block(block_num, proposer.address)

    for address, count in selection_counts.items():
        expected = pos.get_selection_probability(address) * 1000
        print(f'{address}: selected {count} times (expected
        ~{expected:.0f})')
    print("")

    print("SLASHING DEMONSTRATION")
    print("-" * 50)
    slashed = pos.slash_validator("Bob", "double signing")
    print(f'Bob slashed: lost {slashed:.1f} coins')

```

```
print(f"Bob's remaining stake: {pos.validators['Bob'].stake}")  
print(f"Bob can still propose: {not  
pos.validators['Bob'].slashed}")
```

demonstrate_pos()

This simulation shows:

1. Validators are selected proportionally to their stake
2. Over many blocks, selection frequency matches stake percentage
3. Slashing reduces validator stake and can disable them

ADVANTAGES OF PROOF OF STAKE

Energy efficient: No computational puzzles. Uses minimal electricity.

Lower barrier to entry: No specialized hardware required.

Economic alignment: Validators have coins at risk. Attacking the network destroys their own wealth.

Faster finality: Many PoS systems achieve finality in seconds, not hours.

DISADVANTAGES OF PROOF OF STAKE

Nothing at stake problem: In early designs, validators could vote on multiple forks with no penalty. Modern systems solve this with slashing.

Rich get richer: Staking rewards compound. Large stakers earn more, stake more, earn more.

Initial distribution: Someone must get the initial coins. PoW

distributes through mining. PoS inherits whatever distribution existed.

Complexity: PoS protocols are more complex than PoW. More things can go wrong.

Long-range attacks: An attacker with old keys could theoretically create an alternate history. Solved with checkpointing.

SECTION 4: DELEGATED PROOF OF STAKE

WHAT IS DELEGATED PROOF OF STAKE

Delegated Proof of Stake (DPoS) adds a voting layer. Token holders vote for delegates who produce blocks. If a delegate misbehaves, voters can remove them.

EOS, TRON, and Lisk use DPoS.

HOW IT WORKS

1. Token holders vote for block producers (delegates)
2. Votes are weighted by token holdings
3. Top N delegates (often 21 or 101) are selected
4. Delegates take turns producing blocks
5. Delegates share rewards with voters
6. Bad delegates get voted out

PYTHON DPOS SIMULATION

The following code simulates a delegated proof of stake election:

```

class DPoSNetwork:
    def __init__(self, num_delegates = 21):
        self.num_delegates = num_delegates
        self.candidates = {}
        self.voters = {}
        self.active_delegates = []

    def register_candidate(self, address):
        self.candidates[address] = {
            "address": address,
            "votes": 0,
            "voters": [],
            "blocks_produced": 0
        }

    def add_voter(self, address, tokens):
        self.voters[address] = {
            "address": address,
            "tokens": tokens,
            "voted_for": None
        }

    def vote(self, voter_address, candidate_address):
        if voter_address not in self.voters:
            return False
        if candidate_address not in self.candidates:
            return False

        voter = self.voters[voter_address]

        if voter["voted_for"]:
            old_candidate = self.candidates[voter["voted_for"]]
            old_candidate["votes"] -= voter["tokens"]
            old_candidate["voters"].remove(voter_address)

        voter["voted_for"] = candidate_address

```



```
candidate = self.candidates[candidate_address]
candidate["votes"] += voter["tokens"]
candidate["voters"].append(voter_address)
```

```
return True
```

```
def elect_delegates(self):
    sorted_candidates = sorted(
        self.candidates.values(),
        key=lambda c: c["votes"],
        reverse=True
    )
```

```
self.active_delegates =
sorted_candidates[:self.num_delegates]
return self.active_delegates
```

```
def produce_blocks(self, num_blocks):
    if not self.active_delegates:
        self.elect_delegates()
```

```
for i in range(num_blocks):
    delegate_index = i % len(self.active_delegates)
    delegate = self.active_delegates[delegate_index]
    delegate["blocks_produced"] += 1
```

```
def demonstrate_dpos():
    network = DPoSNetwork(num_delegates=5)
```

```
for i in range(10):
    network.register_candidate(f'Delegate_{i}')
```

```
voters_data = [
    ("Whale1", 1000000),
    ("Whale2", 500000),
    ("User1", 10000),
```

```
("User2", 10000),  
("User3", 5000),  
("User4", 5000),  
("User5", 1000),  
]
```

```
for address, tokens in voters_data:  
    network.add_voter(address, tokens)
```

```
network.vote("Whale1", "Delegate_0")  
network.vote("Whale2", "Delegate_1")  
network.vote("User1", "Delegate_2")  
network.vote("User2", "Delegate_2")  
network.vote("User3", "Delegate_3")  
network.vote("User4", "Delegate_4")  
network.vote("User5", "Delegate_5")
```

```
print("ELECTION RESULTS")  
print("-" * 50)
```

```
electd = network.elect_delegates()  
for i, delegate in enumerate(electd):  
    print(f'Rank {i + 1}: {delegate['address']} with  
{delegate['votes']:,} votes")
```

```
print("")  
print("NON-ELECTED CANDIDATES")  
print("-" * 50)
```

```
for candidate in network.candidates.values():  
    if candidate not in electd:  
        print(f'{candidate['address']}: {candidate['votes']:,} votes")
```

```
print("")  
print("BLOCK PRODUCTION (100 blocks)")  
print("-" * 50)
```

```
network.produce_blocks(100)
```

```
for delegate in network.active_delegates:  
    print(f'{delegate['address']}: produced  
{delegate['blocks_produced']} blocks")
```

```
demonstrate_dpos()
```

ADVANTAGES OF DPOS

Very fast: Only a few known delegates. Consensus is quick.

High throughput: Can process thousands of transactions per second.

Democratic: Token holders choose representatives.

Efficient: No wasted computation.

DISADVANTAGES OF DPOS

Centralization: Power concentrates in few delegates.

Plutocracy: Wealthy voters have more influence.

Vote buying: Delegates can bribe voters with rewards.

Cartel formation: Delegates can collude.

SECTION 5: PROOF OF AUTHORITY

WHAT IS PROOF OF AUTHORITY

Proof of Authority (PoA) uses identity as stake. Known, trusted entities validate blocks. Their reputation is at risk, not coins or computing power.

Used in private blockchains, testnets, and some public chains like VeChain.

HOW IT WORKS

1. A set of authorities is defined (by contract, governance, or initial setup)
2. Authorities take turns proposing blocks
3. Other authorities attest to block validity
4. If an authority misbehaves, they are removed
5. Their real-world identity is damaged

PYTHON POA SIMULATION

The following code demonstrates proof of authority consensus:

```
import time
```

```
class PoANetwork:
```

```
    def __init__(self, block_time = 5):
```

```
        self.authorities = {}
```

```
        self.block_time = block_time
```

```
        self.chain = []
```

```
        self.authority_list = []
```

```
    def add_authority(self, address, identity):
```

```
        self.authorities[address] = {
```

```
            "address": address,
```

```
            "identity": identity,
```

```
            "active": True,
```

```

"blocks_signed": 0,
"violations": 0
}
self.authority_list.append(address)

def remove_authority(self, address):
if address in self.authorities:
self.authorities[address]["active"] = False
if address in self.authority_list:
self.authority_list.remove(address)

def get_current_authority(self, block_number):
if not self.authority_list:
return None

index = block_number % len(self.authority_list)
return self.authority_list[index]

def propose_block(self, block_number, proposer_address,
transactions):
expected_authority =
self.get_current_authority(block_number)

if proposer_address != expected_authority:
self.authorities[proposer_address]["violations"] += 1
return None

block = {
"number": block_number,
"proposer": proposer_address,
"transactions": transactions,
"timestamp": int(time.time()),
"attestations": []
}

self.authorities[proposer_address]["blocks_signed"] += 1
return block

```

```
def attest_block(self, block, authority_address):  
    if authority_address not in self.authorities:  
        return False
```

```
    if not self.authorities[authority_address]["active"]:  
        return False
```

```
    if authority_address not in block["attestations"]:  
        block["attestations"].append(authority_address)  
        return True
```

```
    return False
```

```
def finalize_block(self, block):  
    required_attestations = len(self.authority_list) // 2 + 1
```

```
    if len(block["attestations"]) >= required_attestations:  
        self.chain.append(block)  
        return True
```

```
    return False
```

```
def demonstrate_poa():  
    network = PoANetwork()
```

```
    network.add_authority("CompanyA", "Acme Corporation -  
Known entity since 1950")
```

```
    network.add_authority("CompanyB", "GlobalBank - Licensed  
financial institution")
```

```
    network.add_authority("CompanyC", "TechGiant - Fortune  
500 company")
```

```
    network.add_authority("CompanyD", "Government - State  
regulatory body")
```

```
    network.add_authority("CompanyE", "University - Accredited  
research institution")
```

```

print("REGISTERED AUTHORITIES")
print("-" * 60)
for addr, auth in network.authorities.items():
    print(f'{addr}: {auth['identity']}')
print("")

print("BLOCK PRODUCTION")
print("-" * 60)

for block_num in range(10):
    expected = network.get_current_authority(block_num)
    print(f'Block {block_num}: {expected}'s turn to propose")

    block = network.propose_block(block_num, expected,
    [f'tx_{block_num}'])

    for authority in network.authority_list:
        if authority != expected:
            network.attest_block(block, authority)

    if network.finalize_block(block):
        print(f' Finalized with {len(block['attestations'])}
attestations")
    else:
        print(f' Failed to finalize")

    print("")
    print("AUTHORITY STATISTICS")
    print("-" * 60)
    for addr, auth in network.authorities.items():
        print(f'{addr}: {auth['blocks_signed']} blocks signed,
{auth['violations']} violations")

demonstrate_poa()

```

ADVANTAGES OF PROOF OF AUTHORITY

Very fast finality: Known validators can reach consensus quickly.

No token required: Works without cryptocurrency.

High throughput: Can handle many transactions.

Suitable for enterprises: Known identities fit corporate requirements.

DISADVANTAGES OF PROOF OF AUTHORITY

Centralized: Small set of known authorities.

Requires trust: You must trust the authority selection process.

Not censorship resistant: Authorities can collude to censor.

Not suitable for public chains: Identity requirement excludes anonymous participation.

SECTION 6: PRACTICAL BYZANTINE FAULT TOLERANCE

WHAT IS PBFT

Practical Byzantine Fault Tolerance (pBFT) is a consensus algorithm that can tolerate up to one-third of nodes being malicious. It was developed by Miguel Castro and Barbara Liskov in 1999.

Many modern consensus mechanisms are based on pBFT concepts.

HOW IT WORKS

pBFT operates in rounds with four phases:

1. Request: Client sends request to primary node
2. Pre-prepare: Primary broadcasts request to all replicas
3. Prepare: Replicas broadcast prepare messages
4. Commit: Replicas broadcast commit messages

A request is executed when a node receives $2f + 1$ matching commit messages, where f is the maximum number of faulty nodes.

THE MATH

If n = total nodes and f = maximum faulty nodes:

- Need $n \geq 3f + 1$ for safety
- With 4 nodes, can tolerate 1 faulty
- With 7 nodes, can tolerate 2 faulty
- With 10 nodes, can tolerate 3 faulty

PYTHON PBFT SIMULATION

The following code simulates pBFT message passing:

```
from enum import Enum
import hashlib

class MessageType(Enum):
    PRE_PREPARE = 1
    PREPARE = 2
    COMMIT = 3
    REPLY = 4
```

```

class PBFTNode:
    def __init__(self, node_id, total_nodes):
        self.node_id = node_id
        self.total_nodes = total_nodes
        self.f = (total_nodes - 1) // 3
        self.is_primary = (node_id == 0)
        self.prepared = {}
        self.committed = {}
        self.executed = []

    def required_messages(self):
        return 2 * self.f + 1

    def receive_pre_prepare(self, sequence, request):
        request_hash =
        hashlib.sha256(str(request).encode()).hexdigest()[:8]

        self.prepared[sequence] = {
            "request": request,
            "hash": request_hash,
            "prepare_count": 1,
            "commit_count": 0
        }

        return {
            "type": MessageType.PREPARE,
            "node": self.node_id,
            "sequence": sequence,
            "hash": request_hash
        }

    def receive_prepare(self, sequence, sender_hash):
        if sequence not in self.prepared:
            return None

        if self.prepared[sequence]["hash"] == sender_hash:
            self.prepared[sequence]["prepare_count"] += 1

```

```
if self.prepared[sequence]["prepare_count"] > =  
self.required_messages():  
    self.prepared[sequence]["commit_count"] = 1  
    return {  
        "type": MessageType.COMMIT,  
        "node": self.node_id,  
        "sequence": sequence,  
        "hash": sender_hash  
    }
```

```
return None
```

```
def receive_commit(self, sequence, sender_hash):  
    if sequence not in self.prepared:  
        return None
```

```
if self.prepared[sequence]["hash"] == sender_hash:  
    self.prepared[sequence]["commit_count"] += 1
```

```
if self.prepared[sequence]["commit_count"] > =  
self.required_messages():  
    if sequence not in self.executed:  
        self.executed.append(sequence)  
    return {  
        "type": MessageType.REPLY,  
        "node": self.node_id,  
        "sequence": sequence,  
        "result": f"Executed request {sequence}"  
    }
```

```
return None
```

```
class PBFTNetwork:  
    def __init__(self, num_nodes):  
        self.nodes = [PBFTNode(i, num_nodes) for i in
```

```

range(num_nodes)]
self.message_log = []

def process_request(self, request, sequence):
    primary = self.nodes[0]

    print(f"Phase 1: PRIMARY broadcasts PRE-PREPARE")
    pre_prepare_msg = {
        "type": MessageType.PRE_PREPARE,
        "sequence": sequence,
        "request": request
    }
    self.message_log.append(pre_prepare_msg)

    prepare_messages = []
    for node in self.nodes:
        prepare = node.receive_pre_prepare(sequence, request)
        prepare_messages.append(prepare)
        print(f" Node {node.node_id} prepared")

    print(f"Phase 2: Nodes broadcast PREPARE messages")
    commit_messages = []
    for node in self.nodes:
        for prepare in prepare_messages:
            commit = node.receive_prepare(sequence, prepare["hash"])
            if commit and commit not in commit_messages:
                commit_messages.append(commit)

    print(f" {len(commit_messages)} nodes ready to commit")

    print(f"Phase 3: Nodes broadcast COMMIT messages")
    replies = []
    for node in self.nodes:
        for commit in commit_messages:
            reply = node.receive_commit(sequence, commit["hash"])
            if reply:
                replies.append(reply)

```

```

print(f'Phase 4: {len(replies)} nodes executed and replied')

return len(replies) >= self.nodes[0].required_messages()

def demonstrate_pbft():
    print("PBFT CONSENSUS DEMONSTRATION")
    print("=" * 50)
    print("")

    for num_nodes in [4, 7, 10]:
        f = (num_nodes - 1) // 3
        print(f'Network with {num_nodes} nodes (tolerates {f}
        faulty)')
        print("-" * 50)

        network = PBFTNetwork(num_nodes)
        success = network.process_request("Transfer 100 coins",
        sequence = 1)

        print(f'Consensus reached: {success}')
        print("")

    demonstrate_pbft()

```

ADVANTAGES OF PBFT

Immediate finality: Once committed, transactions are final. No need to wait for confirmations.

Energy efficient: No mining or staking required.

Known fault tolerance: Mathematical guarantee of safety with up to f faulty nodes.

DISADVANTAGES OF PBFT

Does not scale: Message complexity is $O(n^2)$. Works for tens of nodes, not thousands.

Known participants: All nodes must know each other. Not permissionless.

Synchrony assumptions: Requires bounded network delays for liveness.

SECTION 7: COMPARING CONSENSUS MECHANISMS

CHOOSING THE RIGHT MECHANISM

Different applications need different properties:

FOR MAXIMUM DECENTRALIZATION AND CENSORSHIP RESISTANCE

Use Proof of Work.

- Anyone can participate
- No identity required
- Hardest to attack or censor
- Highest energy cost

Examples: Bitcoin, Dogecoin

FOR ENERGY EFFICIENCY WITH GOOD DECENTRALIZATION

Use Proof of Stake.

- Much lower energy use
- Still permissionless

- Requires capital instead of hardware
- Faster finality than PoW

Examples: Ethereum, Cardano, Solana, Polkadot

FOR HIGH PERFORMANCE IN CONTROLLED ENVIRONMENTS

Use Proof of Authority or pBFT.

- Fastest consensus
- Lowest resource usage
- Requires trusted participants
- Good for enterprise and private chains

Examples: VeChain, Hyperledger Fabric

FOR COMMUNITY GOVERNANCE

Use Delegated Proof of Stake.

- Democratic element
- High throughput
- Risk of delegate centralization

Examples: EOS, TRON

COMPARISON TABLE

	PoW	PoS	DPoS	PoA	pBFT
Decentralization	High	Medium	Low	Low	Low
Energy Use	Very High	Low	Low	Low	Low
Throughput	Low	Medium	High	High	Medium
Finality Time	Hours	Minutes	Seconds	Seconds	Seconds
Permissionless	Yes	Yes	Yes	No	No
Fault Tolerance	50%	33%	50%	50%	33%

HYBRID APPROACHES

Many modern blockchains combine mechanisms:

Ethereum uses PoS with pBFT-style finality (Casper FFG).

Polkadot uses nominated PoS with GRANDPA finality gadget.

Cosmos uses Tendermint (pBFT-based) with staking.

Solana uses PoS with Proof of History (a timing mechanism).

The trend is toward PoS variants with fast finality, optimized for specific use cases.

SECTION 8: WHAT YOU LEARNED

After reading this chapter, you understand:

1. The consensus problem: how distributed nodes agree without a leader
2. Byzantine Fault Tolerance: the theoretical foundation
3. Proof of Work: computational puzzles, mining, longest chain wins
4. Proof of Stake: economic stake, validator selection, slashing
5. Delegated Proof of Stake: elected delegates, democratic governance
6. Proof of Authority: identity-based, trusted validators
7. pBFT: message-passing consensus with immediate finality

8. Tradeoffs: decentralization vs. throughput vs. finality vs. energy

You have working code that simulates:

- Proof of Work mining with adjustable difficulty
- Proof of Stake validator selection and slashing
- DPoS elections and block production
- Proof of Authority attestation
- pBFT message passing

Understanding consensus is essential. When you write smart contracts, you need to know when your transaction is safe. When you design systems, you need to know how quickly state becomes final. When you choose a blockchain, you need to understand what security model you are accepting.

In the next chapter, we examine transactions and digital signatures: how users prove they authorized a transaction without revealing their private keys.

CHAPTER 3

TRANSACTIONS AND SIGNATURES

A blockchain without transactions is just an empty chain of blocks. Transactions are the purpose. They record value transfers, contract executions, state changes. But how does the network know a transaction is legitimate? How can Alice prove she authorized a transfer without revealing her secret key?

The answer is digital signatures. This chapter explains how they work and how transactions use them.

SECTION 1: WHAT IS A TRANSACTION

THE BASIC CONCEPT

A transaction is a signed message that requests a state change on the blockchain.

In the simplest case, it says: "I, Alice, want to send 5 coins to Bob."

But the network cannot trust this message unless:

1. It can verify Alice actually sent it (authenticity)
2. Alice cannot later deny sending it (non-repudiation)
3. No one can modify it after Alice signed it (integrity)
4. Alice actually has 5 coins to send (validity)

Digital signatures solve the first three. The blockchain state solves the fourth.

ANATOMY OF A TRANSACTION

Different blockchains structure transactions differently, but most contain:

Sender: The address initiating the transaction. Derived from the sender's public key.

Recipient: The address receiving value or being called.

Value: Amount of native currency being transferred.

Data: Additional payload. Empty for simple transfers. Contains function calls for smart contracts.

Nonce: A counter preventing replay attacks. Each transaction from an address has a unique, sequential nonce.

Gas Limit: Maximum computational steps allowed (Ethereum-style chains).

Gas Price: How much the sender pays per computational step.

Signature: Cryptographic proof that the private key holder authorized this transaction.

TRANSACTION EXAMPLE

Here is what an Ethereum transaction looks like in raw form:

[illegible]

The nonce is 42, meaning this is the 43rd transaction from this sender (counting from 0).

The gasPrice is 20 Gwei (20 billion wei).

The gasLimit of 21000 is the exact cost of a simple transfer.

The to address is the recipient.

The value is 1 ETH (1 followed by 18 zeros in wei).

The data is empty because this is a simple transfer, not a contract call.

The v, r, and s fields together form the signature.

SECTION 2: PUBLIC KEY CRYPTOGRAPHY

THE KEY PAIR

Every blockchain address is controlled by a key pair:

Private key: A secret number known only to the owner. Used to sign transactions.

Public key: Derived mathematically from the private key. Can be shared freely. Used to verify signatures.

The critical property: you can derive the public key from the private key, but you cannot derive the private key from the public key. This is a one-way function.

THE RELATIONSHIP

Private key -> Public key -> Address

The private key is a random 256-bit number. In hexadecimal, it looks like:

0x4c0883a69102937d6231471b5dbb6204fe51296170827936ea5cce

The public key is derived using elliptic curve multiplication. It is a point on a specific curve.

The address is derived by hashing the public key and taking

part of the result.

WHY THIS WORKS

Elliptic curve mathematics has a special property: multiplication is easy, but division is practically impossible.

If you know a private key k and a generator point G , computing the public key $K = k * G$ is fast.

But if you only know K and G , finding k requires trying every possible value. With 256 bits, there are more possibilities than atoms in the observable universe.

This is the trapdoor: easy in one direction, impossible in reverse.

PYTHON KEY GENERATION

The following code generates a key pair using the secp256k1 curve (used by Bitcoin and Ethereum):

```
import secrets
import hashlib
```

```
def generate_private_key():
    return secrets.token_hex(32)
```

```
def private_key_to_int(private_key_hex):
    return int(private_key_hex, 16)
```

```
SECP256K1_ORDER =
```

```
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD2
```

```
SECP256K1_GX =
```

```
0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959
```

```
SECP256K1_GY =
```

```
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C
SECP256K1_P =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
def mod_inverse(a, m):
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y
```

```
_, x, _ = extended_gcd(a % m, m)
return (x % m + m) % m
```

```
def point_add(p1, p2):
    if p1 is None:
        return p2
    if p2 is None:
        return p1
```

```
x1, y1 = p1
x2, y2 = p2
```

```
if x1 == x2 and y1 != y2:
    return None
```

```
if x1 == x2:
    m = (3 * x1 * x1) * mod_inverse(2 * y1, SECP256K1_P) %
SECP256K1_P
    else:
        m = (y2 - y1) * mod_inverse(x2 - x1, SECP256K1_P) %
SECP256K1_P
```

```
x3 = (m * m - x1 - x2) % SECP256K1_P
y3 = (m * (x1 - x3) - y1) % SECP256K1_P
```

```
return (x3, y3)
```

```
def scalar_multiply(k, point):
```

```
    result = None
```

```
    addend = point
```

```
    while k:
```

```
        if k & 1:
```

```
            result = point_add(result, addend)
```

```
            addend = point_add(addend, addend)
```

```
            k >>= 1
```

```
    return result
```

```
def private_to_public(private_key_hex):
```

```
    private_int = private_key_to_int(private_key_hex)
```

```
    generator = (SECP256K1_GX, SECP256K1_GY)
```

```
    public_point = scalar_multiply(private_int, generator)
```

```
    return public_point
```

```
def public_key_to_address(public_point):
```

```
    x, y = public_point
```

```
    public_bytes = x.to_bytes(32, 'big') + y.to_bytes(32, 'big')
```

```
    keccak = hashlib.sha3_256(public_bytes).digest()
```

```
    address = '0x' + keccak[-20:].hex()
```

```
    return address
```

```
def demonstrate_keys():
```

```
    private_key = generate_private_key()
```

```
    print(f"Private Key: 0x{private_key}")
```

```
    public_point = private_to_public(private_key)
```

```
    print(f"Public Key X: {hex(public_point[0])}")
```

```
    print(f"Public Key Y: {hex(public_point[1])}")
```

```
address = public_key_to_address(public_point)
print(f'Address: {address}')
```

```
demonstrate_keys()
```

This code implements elliptic curve operations from scratch. In production, you would use a library like `ecdsa` or `eth-keys`, but understanding the math matters.

SECTION 3: DIGITAL SIGNATURES

WHAT IS A DIGITAL SIGNATURE

A digital signature is a mathematical proof that:

1. A specific private key was used to create it
2. It was created for a specific message
3. The message has not been modified

Anyone with the public key can verify the signature. But only the private key holder can create it.

THE SIGNING PROCESS

To sign a message:

1. Hash the message to get a fixed-size digest
2. Use the private key and digest to compute a signature
3. The signature consists of two numbers: r and s

To verify a signature:

1. Hash the message to get the same digest
2. Use the public key, digest, and signature to verify

3. The math confirms whether the signature is valid

WHY SIGNATURES CANNOT BE FORGED

To forge a signature, an attacker would need to:

1. Know the private key (impossible to derive from public key)
2. Or find a mathematical shortcut (none exists for ECDSA with current knowledge)
3. Or find a hash collision (computationally infeasible)

The security comes from multiple hard problems combined.

ECDSA: ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM

ECDSA is the signature algorithm used by Bitcoin, Ethereum, and many other blockchains.

Signing with ECDSA:

1. Generate a random number k (the ephemeral key)
2. Compute point $R = k * G$ (where G is the generator point)
3. Set $r = R.x \bmod n$ (the x-coordinate of R , modulo the curve order)
4. Compute $s = k^{-1} * (\text{hash} + r * \text{privateKey}) \bmod n$
5. The signature is (r, s)

Verification with ECDSA:

1. Compute $u1 = \text{hash} * s^{-1} \bmod n$
2. Compute $u2 = r * s^{-1} \bmod n$
3. Compute point $P = u1 * G + u2 * \text{PublicKey}$
4. If $P.x \bmod n$ equals r , the signature is valid

PYTHON ECDSA IMPLEMENTATION

The following code implements ECDSA signing and verification:

```
import secrets
import hashlib

class ECDSA:
    def __init__(self):
        self.p = SECP256K1_P
        self.n = SECP256K1_ORDER
        self.G = (SECP256K1_GX, SECP256K1_GY)

    def hash_message(self, message):
        if isinstance(message, str):
            message = message.encode()
        return int(hashlib.sha256(message).hexdigest(), 16)

    def sign(self, private_key_hex, message):
        private_key = int(private_key_hex, 16)
        z = self.hash_message(message)

        while True:
            k = secrets.randbelow(self.n - 1) + 1

            R = scalar_multiply(k, self.G)
            r = R[0] % self.n

            if r == 0:
                continue

            k_inv = mod_inverse(k, self.n)
            s = (k_inv * (z + r * private_key)) % self.n

            if s == 0:
                continue

        return (r, s)
```

```
def verify(self, public_point, message, signature):  
    r, s = signature
```

```
    if not (1 <= r < self.n and 1 <= s < self.n):  
        return False
```

```
    z = self.hash_message(message)
```

```
    s_inv = mod_inverse(s, self.n)
```

```
    u1 = (z * s_inv) % self.n
```

```
    u2 = (r * s_inv) % self.n
```

```
    point1 = scalar_multiply(u1, self.G)
```

```
    point2 = scalar_multiply(u2, public_point)
```

```
    P = point_add(point1, point2)
```

```
    if P is None:
```

```
        return False
```

```
    return P[0] % self.n == r
```

```
def recover_public_key(self, message, signature, recovery_id):
```

```
    r, s = signature
```

```
    z = self.hash_message(message)
```

```
    x = r
```

```
    y_squared = (pow(x, 3, self.p) + 7) % self.p
```

```
    y = pow(y_squared, (self.p + 1) // 4, self.p)
```

```
    if recovery_id % 2 != y % 2:
```

```
        y = self.p - y
```

```
    R = (x, y)
```

```
    r_inv = mod_inverse(r, self.n)
```

```
    u1 = (-z * r_inv) % self.n
```

```
u2 = (s * r_inv) % self.n
```

```
point1 = scalar_multiply(u1, self.G)
```

```
point2 = scalar_multiply(u2, R)
```

```
public_key = point_add(point1, point2)
```

```
return public_key
```

```
def demonstrate_ecdsa():
```

```
    ecdsa = ECDSA()
```

```
    private_key = generate_private_key()
```

```
    public_key = private_to_public(private_key)
```

```
    print("KEY PAIR GENERATED")
```

```
    print(f'Private: 0x{private_key[:16]}...')
```

```
    print(f'Public X: {hex(public_key[0]):20}...')
```

```
    print("")
```

```
    message = "Transfer 100 coins from Alice to Bob"
```

```
    signature = ecdsa.sign(private_key, message)
```

```
    print("MESSAGE SIGNED")
```

```
    print(f'Message: {message}')
```

```
    print(f'Signature r: {hex(signature[0]):20}...')
```

```
    print(f'Signature s: {hex(signature[1]):20}...')
```

```
    print("")
```

```
    is_valid = ecdsa.verify(public_key, message, signature)
```

```
    print(f'Signature valid: {is_valid}')
```

```
    print("")
```

```
    tampered_message = "Transfer 1000 coins from Alice to Bob"
```

```
    is_valid_tampered = ecdsa.verify(public_key,  
    tampered_message, signature)
```

```
    print(f'Tampered message valid: {is_valid_tampered}')
```

```
print("")
```

```
wrong_key = private_to_public(generate_private_key())  
is_valid_wrong_key = ecdsa.verify(wrong_key, message,  
signature)
```

```
print(f"Wrong key valid: {is_valid_wrong_key}")
```

```
demonstrate_ecdsa()
```

This demonstrates:

1. Signing a message produces a unique signature
2. The correct public key verifies the signature
3. Tampering with the message invalidates the signature
4. A different public key cannot verify the signature

SECTION 4: EdDSA AND ED25519

WHAT IS EdDSA

EdDSA (Edwards-curve Digital Signature Algorithm) is a newer signature scheme using twisted Edwards curves. Ed25519 is the most common variant, using Curve25519.

Solana, Cardano, and many newer blockchains prefer Ed25519 over ECDSA.

WHY EdDSA OVER ECDSA

Faster: Ed25519 operations are significantly faster.

Simpler: The algorithm is cleaner and easier to implement correctly.

Deterministic: No random number needed during signing. Same message always produces same signature. This eliminates a class of vulnerabilities.

Safer: Designed to be resistant to side-channel attacks.

Smaller signatures: 64 bytes instead of 70-72 for ECDSA.

THE ED25519 ALGORITHM

Ed25519 uses the equation: $-x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2$

The base point and curve parameters are fixed by the standard.

Signing:

1. Hash private key to get two 32-byte values: a (scalar) and prefix
2. Hash prefix + message to get r (deterministic, no random needed)
3. Compute $R = r * \text{BasePoint}$
4. Compute $k = \text{Hash}(R || \text{PublicKey} || \text{message})$
5. Compute $s = r + k * a \text{ mod order}$
6. Signature is (R, s) - 64 bytes total

Verification:

1. Compute $k = \text{Hash}(R || \text{PublicKey} || \text{message})$
2. Check: $s * \text{BasePoint} == R + k * \text{PublicKey}$

USING ED25519 IN PYTHON

The following code uses the nacl library for Ed25519 operations:

```
from nacl.signing import SigningKey, VerifyKey
from nacl.encoding import HexEncoder
```

```
import nacl.exceptions

class Ed25519Wallet:
    def __init__(self, private_key_bytes = None):
        if private_key_bytes:
            self.signing_key = SigningKey(private_key_bytes)
        else:
            self.signing_key = SigningKey.generate()

        self.verify_key = self.signing_key.verify_key

    def get_private_key(self):
        return
        self.signing_key.encode(encoder = HexEncoder).decode()

    def get_public_key(self):
        return self.verify_key.encode(encoder = HexEncoder).decode()

    def sign(self, message):
        if isinstance(message, str):
            message = message.encode()

        signed = self.signing_key.sign(message)
        return signed.signature.hex()

    def verify(self, message, signature_hex,
               public_key_hex = None):
        if isinstance(message, str):
            message = message.encode()

        signature = bytes.fromhex(signature_hex)

        if public_key_hex:
            verify_key = VerifyKey(bytes.fromhex(public_key_hex))
        else:
            verify_key = self.verify_key
```

```
try:
    verify_key.verify(message, signature)
    return True
except nacl.exceptions.BadSignature:
    return False
```

```
def demonstrate_ed25519():
    print("ED25519 DEMONSTRATION")
    print("=" * 50)
    print("")
```

```
wallet = Ed25519Wallet()
```

```
print("KEYS GENERATED")
print(f"Private: {wallet.get_private_key()[:32]}...")
print(f"Public: {wallet.get_public_key()}")
print("")
```

```
message = "Send 50 SOL to recipient address"
signature = wallet.sign(message)
```

```
print("MESSAGE SIGNED")
print(f"Message: {message}")
print(f"Signature: {signature[:32]}...")
print(f"Signature length: {len(signature) // 2} bytes")
print("")
```

```
is_valid = wallet.verify(message, signature)
print(f"Valid signature: {is_valid}")
```

```
tampered = "Send 500 SOL to recipient address"
is_valid_tampered = wallet.verify(tampered, signature)
print(f"Tampered message: {is_valid_tampered}")
```

```
other_wallet = Ed25519Wallet()
is_valid_wrong = wallet.verify(message, signature,
```



```
other_wallet.get_public_key()  
print(f"Wrong public key: {is_valid_wrong}")  
  
demonstrate_ed25519()
```

Note: This code requires the PyNaCl library. Install with: `pip install pynacl`

COMPARISON: ECDSA VS ED25519

	ECDSA (secp256k1)	Ed25519
Curve Type	Weierstrass	Twisted Edwards
Key Size	256 bits	256 bits
Signature Size	70-72 bytes	64 bytes
Signing Speed	Slower	Faster
Verification	Slower	Faster
Deterministic	No (needs random k)	Yes
Used By	Bitcoin, Ethereum	Solana, Cardano

Both are secure. Ed25519 is generally preferred for new systems due to its better properties.

SECTION 5: NONCES AND REPLAY PROTECTION

THE REPLAY ATTACK PROBLEM

Suppose Alice signs a transaction: "Send 10 coins to Bob."

Without protection, an attacker could:

1. Capture the signed transaction
2. Submit it again
3. Alice loses another 10 coins
4. Repeat indefinitely

This is a replay attack. The signature is valid every time because Alice did sign that message.

THE NONCE SOLUTION

A nonce is a number used once. Each account has a nonce counter. Each transaction must include the next expected nonce.

Alice's nonce starts at 0.

- Transaction 1: nonce 0, "Send 10 to Bob"
- Transaction 2: nonce 1, "Send 5 to Carol"
- Transaction 3: nonce 2, "Send 3 to Dave"

If someone replays transaction 1:

- The network expects nonce 3
- The replayed transaction has nonce 0
- Rejected: nonce too low

If someone tries to submit nonce 5:

- The network expects nonce 3
- Rejected: nonce too high (gap not allowed in most chains)

NONCE IMPLEMENTATION

The following code demonstrates nonce tracking:

```
class NonceManager:
```

```
    def __init__(self):
```

```
        self.nonces = {}
```

```
    def get_current_nonce(self, address):
```

```
        return self.nonces.get(address, 0)
```

```
    def validate_and_increment(self, address, provided_nonce):
```

```
expected = self.get_current_nonce(address)
```

```
if provided_nonce < expected:  
    return False, f"Nonce too low: got {provided_nonce},  
    expected {expected}"
```

```
if provided_nonce > expected:  
    return False, f"Nonce too high: got {provided_nonce},  
    expected {expected}"
```

```
self.nonces[address] = expected + 1  
return True, "Nonce valid"
```

```
class TransactionPool:
```

```
    def __init__(self):  
        self.nonce_manager = NonceManager()  
        self.pending = []  
        self.processed = []
```

```
    def submit_transaction(self, tx):  
        address = tx["from"]  
        nonce = tx["nonce"]
```

```
        valid, message =  
        self.nonce_manager.validate_and_increment(address, nonce)
```

```
        if not valid:  
            return False, message
```

```
        self.pending.append(tx)  
        return True, "Transaction accepted"
```

```
    def process_pending(self):  
        while self.pending:  
            tx = self.pending.pop(0)  
            self.processed.append(tx)
```

```
def demonstrate_nonce():
    pool = TransactionPool()

    print("NONCE DEMONSTRATION")
    print("=" * 50)
    print("")

    tx1 = {"from": "Alice", "to": "Bob", "amount": 10, "nonce": 0}
    success, msg = pool.submit_transaction(tx1)
    print(f"TX1 (nonce 0): {msg}")

    tx2 = {"from": "Alice", "to": "Carol", "amount": 5, "nonce": 1}
    success, msg = pool.submit_transaction(tx2)
    print(f"TX2 (nonce 1): {msg}")

    print("")
    print("REPLAY ATTACK ATTEMPT")

    replay = {"from": "Alice", "to": "Bob", "amount": 10, "nonce":
0}
    success, msg = pool.submit_transaction(replay)
    print(f"Replay TX1 (nonce 0): {msg}")

    print("")
    print("SKIP NONCE ATTEMPT")

    skip = {"from": "Alice", "to": "Dave", "amount": 7, "nonce": 5}
    success, msg = pool.submit_transaction(skip)
    print(f"Skip to nonce 5: {msg}")

    print("")
    print("CORRECT NEXT TRANSACTION")

    tx3 = {"from": "Alice", "to": "Eve", "amount": 3, "nonce": 2}
    success, msg = pool.submit_transaction(tx3)
    print(f"TX3 (nonce 2): {msg}")
```

demonstrate_nonce()

CROSS-CHAIN REPLAY PROTECTION

When Ethereum forked into ETH and ETC in 2016, transactions were valid on both chains. People could replay transactions across chains.

Modern transactions include a chain ID in the signed data. A transaction for chain ID 1 (Ethereum mainnet) is invalid on chain ID 56 (BNB Chain).

EIP-155 standardized this for Ethereum:

```
transaction_data = rlp_encode([
    nonce,
    gas_price,
    gas_limit,
    to,
    value,
    data,
    chain_id,
    0,
    0
])
```

```
signature = sign(keccak256(transaction_data))
```

The chain ID becomes part of the signed message, making cross-chain replay impossible.

SECTION 6: TRANSACTION LIFECYCLE

STEP 1: CREATION

The user creates a transaction object:

- Specifies recipient, amount, data
- Gets current nonce from network
- Estimates gas requirements
- Sets gas price based on network conditions

STEP 2: SIGNING

The wallet:

- Serializes the transaction
- Hashes the serialized data
- Signs the hash with the private key
- Attaches signature to transaction

STEP 3: BROADCAST

The signed transaction is sent to a node:

- Via JSON-RPC API
- Via WebSocket connection
- Via peer-to-peer gossip

The node performs initial validation:

- Is the signature valid?
- Does the sender have enough balance?
- Is the nonce correct?
- Is the gas price acceptable?

STEP 4: MEMPOOL

Valid transactions enter the mempool (memory pool):

- Waiting area for unconfirmed transactions
- Nodes share mempool contents
- Miners/validators select transactions to include

STEP 5: INCLUSION

A block producer:

- Selects transactions from mempool (usually highest gas price first)
- Executes transactions in order
- Builds a block containing the transactions
- Mines or validates the block

STEP 6: CONFIRMATION

The block is added to the chain:

- Other nodes verify and accept the block
- Transaction moves from mempool to blockchain
- State changes become part of chain history

STEP 7: FINALITY

After enough confirmations:

- Transaction is considered irreversible
- 1 confirmation: block included (might be reorganized)
- 6 confirmations (Bitcoin): practically final
- 32 slots (Ethereum PoS): finalized by protocol

PYTHON TRANSACTION LIFECYCLE SIMULATION

The following code simulates the complete transaction lifecycle:

```
import time
import hashlib
import json
from enum import Enum
```

```
class TxStatus(Enum):
    CREATED = "created"
    SIGNED = "signed"
    PENDING = "pending"
    INCLUDED = "included"
    CONFIRMED = "confirmed"
    FINALIZED = "finalized"
```

```
class Transaction:
    def __init__(self, sender, recipient, amount, nonce,
chain_id=1):
    self.sender = sender
    self.recipient = recipient
    self.amount = amount
    self.nonce = nonce
    self.chain_id = chain_id
    self.gas_limit = 21000
    self.gas_price = 20
    self.data = ""
    self.signature = None
    self.hash = None
    self.status = TxStatus.CREATED
    self.block_number = None
    self.confirmations = 0
```

```
def to_dict(self):
    return {
        "sender": self.sender,
        "recipient": self.recipient,
        "amount": self.amount,
        "nonce": self.nonce,
        "chain_id": self.chain_id,
        "gas_limit": self.gas_limit,
        "gas_price": self.gas_price,
        "data": self.data
    }
```



```
def get_signing_hash(self):
    data = json.dumps(self.to_dict(), sort_keys=True)
    return hashlib.sha256(data.encode()).hexdigest()
```

```
def sign(self, private_key):
    signing_hash = self.get_signing_hash()
    self.signature = hashlib.sha256(f'{private_key}
{signing_hash}'.encode()).hexdigest()
    self.hash = hashlib.sha256(f'{self.signature}
{signing_hash}'.encode()).hexdigest()
    self.status = TxStatus.SIGNED
    return self.hash
```

```
class Mempool:
    def __init__(self):
        self.transactions = {}
```

```
def add(self, tx):
    if tx.status != TxStatus.SIGNED:
        return False, "Transaction not signed"
```

```
if tx.hash in self.transactions:
    return False, "Transaction already in mempool"
```

```
self.transactions[tx.hash] = tx
tx.status = TxStatus.PENDING
return True, "Added to mempool"
```

```
def get_transactions(self, limit=10):
    sorted_txs = sorted(
        self.transactions.values(),
        key=lambda t: t.gas_price,
        reverse=True
    )
    return sorted_txs[:limit]
```

```
def remove(self, tx_hash):
    if tx_hash in self.transactions:
        del self.transactions[tx_hash]
```

```
class Blockchain:
    def __init__(self):
        self.blocks = []
        self.mempool = Mempool()
        self.balances = {"Alice": 1000, "Bob": 500, "Carol": 250}
        self.nonces = {"Alice": 0, "Bob": 0, "Carol": 0}
```

```
def submit_transaction(self, tx):
    if tx.sender not in self.balances:
        return False, "Unknown sender"
```

```
    if self.balances[tx.sender] < tx.amount + (tx.gas_limit *
tx.gas_price):
        return False, "Insufficient balance"
```

```
    if tx.nonce != self.nonces.get(tx.sender, 0):
        return False, f"Invalid nonce: expected
{self.nonces.get(tx.sender, 0)}"
```

```
    return self.mempool.add(tx)
```

```
def mine_block(self):
    txs = self.mempool.get_transactions(5)
    if not txs:
        return None
```

```
    block_number = len(self.blocks)
    block = {
        "number": block_number,
        "transactions": [],
        "timestamp": int(time.time())
    }
```

```

for tx in txs:
    self.balances[tx.sender] -= tx.amount
    self.balances[tx.recipient] = self.balances.get(tx.recipient, 0)
    + tx.amount
    self.nonces[tx.sender] = tx.nonce + 1

tx.status = TxStatus.INCLUDED
tx.block_number = block_number
tx.confirmations = 1

block["transactions"].append(tx.hash)
self.mempool.remove(tx.hash)

self.blocks.append(block)
return block

def add_confirmations(self):
    for block in self.blocks:
        for tx_hash in block["transactions"]:
            pass

def demonstrate_lifecycle():
    print("TRANSACTION LIFECYCLE DEMONSTRATION")
    print("=" * 60)
    print("")

    blockchain = Blockchain()

    print("INITIAL BALANCES")
    for addr, bal in blockchain.balances.items():
        print(f" {addr}: {bal}")
    print("")

    print("STEP 1: CREATE TRANSACTION")
    tx = Transaction(
        sender="Alice",

```

```
recipient = "Bob",
amount = 100,
nonce = 0
)
print(f" Status: {tx.status.value}")
print(f" From: {tx.sender} To: {tx.recipient} Amount:
{tx.amount}")
print("")
```

```
print("STEP 2: SIGN TRANSACTION")
tx_hash = tx.sign("alice_private_key_12345")
print(f" Status: {tx.status.value}")
print(f" Hash: {tx_hash[:20]}...")
print(f" Signature: {tx.signature[:20]}...")
print("")
```

```
print("STEP 3: BROADCAST TO MEMPOOL")
success, msg = blockchain.submit_transaction(tx)
print(f" Result: {msg}")
print(f" Status: {tx.status.value}")
print(f" Mempool size:
{len(blockchain.mempool.transactions)})")
print("")
```

```
tx2 = Transaction(sender = "Bob", recipient = "Carol",
amount = 50, nonce = 0)
tx2.sign("bob_private_key_67890")
blockchain.submit_transaction(tx2)
print(f" Added second transaction (Bob -> Carol)")
print(f" Mempool size:
{len(blockchain.mempool.transactions)})")
print("")
```

```
print("STEP 4: MINE BLOCK")
block = blockchain.mine_block()
print(f" Block {block['number']} mined")
print(f" Transactions included: {len(block['transactions'])}")
```

```
print(f" TX1 Status: {tx.status.value}")
print(f" TX1 Block: {tx.block_number}")
print(f" TX1 Confirmations: {tx.confirmations}")
print("")
```

```
print("FINAL BALANCES")
for addr, bal in blockchain.balances.items():
    print(f" {addr}: {bal}")
print("")
```

```
print("REPLAY ATTACK TEST")
replay_tx = Transaction(sender="Alice", recipient="Bob",
amount=100, nonce=0)
replay_tx.sign("alice_private_key_12345")
success, msg = blockchain.submit_transaction(replay_tx)
print(f" Replay attempt: {msg}")
```

```
demonstrate_lifecycle()
```

SECTION 7: TRANSACTION TYPES

SIMPLE TRANSFERS

The most basic transaction: send native currency from one address to another.

```
{
  "to": "0xRecipient...",
  "value": 1000000000000000000,
  "data": ""
}
```

CONTRACT CREATION

When "to" is empty or null, a new contract is deployed. The "data" field contains the contract bytecode.

```
{
  "to": null,
  "value": 0,
  "data": "0x608060405234801561001057600080fd5b50..."
}
```

CONTRACT CALLS

Calling a function on an existing contract. The "data" field contains the encoded function call.

```
{
  "to": "0xContractAddress...",
  "value": 0,
  "data": "0xa9059cbb000000000000000000000000recipient..."
}
```

The first 4 bytes (0xa9059cbb) are the function selector: keccak256("transfer(address,uint256)")[4]

The remaining bytes are the encoded parameters.

EIP-1559 TRANSACTIONS

Ethereum's fee market change introduced new transaction fields:

```
{
  "type": 2,
  "maxFeePerGas": 100000000000,
  "maxPriorityFeePerGas": 2000000000,
  "to": "0xRecipient...",
  "value": 1000000000000000000,

```

```
"data": ""  
}
```

Instead of a single gas price, users specify:

- maxFeePerGas: Maximum total fee willing to pay
- maxPriorityFeePerGas: Tip for the block producer

The actual fee is: $\min(\text{maxFeePerGas}, \text{baseFee} + \text{maxPriorityFeePerGas})$

The base fee is burned. The priority fee goes to the producer.

SECTION 8: WHAT YOU LEARNED

After reading this chapter, you understand:

1. Transaction structure: sender, recipient, value, data, nonce, gas, signature
2. Public key cryptography: private keys, public keys, addresses, one-way derivation
3. Digital signatures: how they prove authorization without revealing secrets
4. ECDSA: the algorithm used by Bitcoin and Ethereum, including the math
5. Ed25519: the faster, simpler alternative used by Solana and others
6. Nonces: how they prevent replay attacks
7. Chain IDs: how they prevent cross-chain replay
8. Transaction lifecycle: creation, signing, broadcast,

mempool, inclusion, finality

You have working code that:

- Generates key pairs using elliptic curve math
- Signs and verifies messages with ECDSA
- Uses Ed25519 for faster signatures
- Tracks nonces to prevent replay
- Simulates the complete transaction lifecycle

This cryptographic foundation enables everything else. Smart contracts are just transactions with code. DeFi is transactions interacting with protocols. NFTs are transactions creating tokens. Every blockchain action is a signed transaction.

In the next chapter, we explore major blockchain platforms: Ethereum, Solana, Bitcoin, Polygon, and others. You will learn their architectures, tradeoffs, and how to choose the right one for your project.

CHAPTER 4

MAJOR BLOCKCHAIN PLATFORMS

You understand how blockchains work. You understand consensus. You understand transactions and signatures. Now the question is: which blockchain should you build on?

This is not a simple question. Each platform makes different tradeoffs. The right choice depends on what you are building, who your users are, and what properties matter most.

This chapter examines the major platforms: their architectures, their tradeoffs, their ecosystems, and their futures.

SECTION 1: THE LANDSCAPE IN 2025-2026

THE CURRENT STATE

The blockchain landscape has matured significantly. As of 2025, the mindshare distribution looks like this:

Solana: 26.79% of global mindshare

Base: 13.94%

Ethereum: 13.43%

Sui: 11.77%

BNB Chain: 9.05%

Solana leads in raw usage with 98 million monthly active users and 34 billion transactions. Over 3.6 million unique wallet addresses are active daily on Solana -- nearly seven times more than Ethereum's 530,000 daily active addresses.

But Ethereum leads in Total Value Locked (TVL), developer activity, and stablecoin supply.

These numbers tell a story: different platforms serve different purposes.

THE THREE CATEGORIES

Layer 1 blockchains fall into three broad categories:

Maximum Decentralization: Bitcoin, Ethereum

- Thousands of nodes worldwide
- Slower but battle-tested
- Highest security guarantees
- Higher transaction costs

Maximum Performance: Solana, Sui, Aptos

- Fewer but powerful nodes
- Thousands of transactions per second

- Sub-second finality
- Very low fees

EVM Compatibility: Polygon, Arbitrum, Base, BNB Chain

- Compatible with Ethereum tooling
- Faster and cheaper than Ethereum mainnet
- Varying degrees of decentralization
- Easy migration path for Ethereum developers

CHOOSING A PLATFORM

Ask yourself:

What matters most -- security or speed?

If your application handles billions of dollars, security matters more. If you are building a game or social app, speed matters more.

Who are your users?

Retail users want low fees and fast confirmations. Institutions want security and regulatory clarity.

What ecosystem do you need?

DeFi is concentrated on Ethereum and its L2s. NFT trading is active on both Ethereum and Solana. Gaming favors Solana, Sui, and Polygon.

What is your team's experience?

Solidity developers should consider EVM chains. Rust developers might prefer Solana.

SECTION 2: ETHEREUM

WHAT IS ETHEREUM

Ethereum is the first blockchain to support smart contracts -- programs that run on the blockchain. Launched in 2015 by Vitalik Buterin and others, Ethereum created the foundation for DeFi, NFTs, and decentralized applications.

THE ETHEREUM VIRTUAL MACHINE

The EVM (Ethereum Virtual Machine) is the runtime environment for smart contracts. Every node runs the same EVM, executing the same code, producing the same results.

Key properties:

Deterministic: Same input always produces same output

Isolated: Contracts cannot access external resources

Metered: Every operation costs gas, preventing infinite loops

Stack-based: 256-bit words, 1024 stack depth

ETHEREUM'S ARCHITECTURE

Execution Layer (formerly Eth1):

- Processes transactions
- Executes smart contracts
- Maintains state (account balances, contract storage)

Consensus Layer (formerly Eth2):

- Proof of Stake consensus (since September 2022)
- Validators stake 32 ETH
- Finality in about 12 minutes (2 epochs)

TRANSACTION COSTS

Ethereum mainnet is expensive. A simple transfer costs around 21,000 gas. At 30 gwei gas price, that is about

0.00063 ETH. At \$3000/ETH, that is roughly \$2.

Smart contract interactions cost more. A Uniswap swap might cost 150,000 gas -- around \$15. During network congestion, costs spike to hundreds of dollars.

This is why Layer 2 solutions exist.

THE LAYER 2 STRATEGY

Ethereum's scaling strategy is to move execution to Layer 2 rollups while keeping settlement on Layer 1.

Optimistic Rollups (Arbitrum, Optimism, Base):

- Assume transactions are valid
- Fraud proofs if someone cheats
- 7-day withdrawal delay

ZK Rollups (zkSync, StarkNet, Polygon zkEVM):

- Prove validity with zero-knowledge proofs
- Faster withdrawals
- More complex to build

CONNECTING TO ETHEREUM

Here is how to connect to Ethereum using Python:

```
from web3 import Web3
```

```
infura_url = "https://mainnet.infura.io/v3/  
YOUR_PROJECT_ID"
```

```
w3 = Web3(Web3.HTTPProvider(infura_url))
```

```
print(f'Connected: {w3.is_connected()}')
```

```
print(f'Block number: {w3.eth.block_number}')
```

```
print(f'Gas price: {w3.eth.gas_price / 10**9} gwei')
```

```
address =  
"0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045"  
balance = w3.eth.get_balance(address)  
print(f'Balance: {w3.from_wei(balance, 'ether')} ETH')
```

WHEN TO USE ETHEREUM

Use Ethereum mainnet when:

- Security is paramount (large value at stake)
- You need maximum decentralization
- Your users are institutions or whales
- You are building core DeFi infrastructure

Use Ethereum L2s when:

- You want Ethereum security with lower costs
- Your application needs more throughput
- You are targeting retail users
- You want EVM compatibility

ETHEREUM STRENGTHS

Battle-tested: Zero downtime over nearly a decade

Developer ecosystem: Most tools, libraries, documentation

DeFi dominance: Highest TVL, most liquidity

Institutional trust: ETH ETFs approved, regulatory clarity

Network effects: Most developers, most users, most applications

ETHEREUM WEAKNESSES

High costs: Mainnet transactions are expensive

Slow finality: 12+ minutes for true finality

Fragmented liquidity: L2s split the ecosystem

Complexity: The L2 landscape is confusing for new users

SECTION 3: SOLANA

WHAT IS SOLANA

Solana is a high-performance blockchain designed for speed. Launched in 2020 by Anatoly Yakovenko, Solana can process thousands of transactions per second with sub-second finality.

Where Ethereum chose decentralization first, Solana chose performance first.

PROOF OF HISTORY

Solana's key innovation is Proof of History (PoH) -- a cryptographic clock that timestamps transactions before consensus.

In traditional blockchains, nodes must agree on the order of transactions. This communication takes time.

With PoH, transactions are stamped with a verifiable order before they reach consensus. Validators can process transactions in parallel because the order is already determined.

The PoH generator continuously hashes:

hash1 = sha256(previous_hash)

hash2 = sha256(hash1)

hash3 = sha256(hash2)

...

Each hash proves that time passed. Transactions inserted between hashes have a verifiable timestamp.

SOLANA'S ARCHITECTURE

Unlike Ethereum's account model, Solana uses an account model with a twist: programs (smart contracts) and data are stored separately.

Programs: Executable code, stored in program accounts

Accounts: Data storage, owned by programs

Transactions: Instructions that invoke programs with accounts

This separation allows programs to be upgraded without redeploying all data.

PERFORMANCE NUMBERS

Transaction throughput: Up to 65,000 TPS theoretical,
2,000-3,000 TPS typical

Finality: 400 milliseconds

Block time: 400 milliseconds

Transaction cost: \$0.00025 average

THE VALIDATOR REQUIREMENTS

Solana validators need serious hardware:

- 128 GB RAM minimum (256 GB recommended)
- 12-core CPU or better
- 2 TB NVMe SSD
- 1 Gbps network connection

This limits who can run validators, reducing decentralization. About 1,900 validators secure the network, compared to Ethereum's hundreds of thousands of stakers.

CONNECTING TO SOLANA

Here is how to connect to Solana using Python:

```
from solana.rpc.api import Client
from solders.pubkey import Pubkey
```

```
client = Client("https://api.mainnet-beta.solana.com")
```

```
response = client.get_slot()
print(f"Current slot: {response.value}")
```

```
response = client.get_block_height()
print(f"Block height: {response.value}")
```

```
address =  
Pubkey.from_string("Vote1111111111111111111111111111111111111111")  
response = client.get_balance(address)  
print(f'Balance: {response.value / 10**9} SOL')
```

```
response = client.get_recent_blockhash()
print(f'Recent blockhash: {response.value.blockhash}')
```

PROGRAMMING ON SOLANA

Solana programs are written in Rust (or C). The Anchor framework simplifies development:

```
use anchor_lang::prelude::*;
```

```
declare_id!  
("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");
```

```
#[program]
pub mod basic_program {
  use super::*;
```



```

pub fn initialize(ctx: Context<Initialize>, data: u64) ->
Result<()> {
    let my_account = &mut ctx.accounts.my_account;
    my_account.data = data;
    Ok(())
}

```

```

pub fn update(ctx: Context<Update>, data: u64) ->
Result<()> {
    let my_account = &mut ctx.accounts.my_account;
    my_account.data = data;
    Ok(())
}
}

```

```

#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(init, payer = user, space = 8 + 8)]
    pub my_account: Account<'info, MyAccount>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

```

```

#[derive(Accounts)]
pub struct Update<'info> {
    #[account(mut)]
    pub my_account: Account<'info, MyAccount>,
}

```

```

#[account]
pub struct MyAccount {
    pub data: u64,
}

```

FIREDANCER: THE 2026 UPGRADE

Firedancer is a new validator client being built by Jump Crypto. It rewrites the Solana validator from scratch in C/C++ with extreme performance optimizations.

Expected improvements:

- 1 million+ TPS theoretical
- Better stability and reliability
- Reduced hardware requirements
- Client diversity (reducing single-client risk)

Firedancer is expected to fully launch in 2026.

WHEN TO USE SOLANA

Use Solana when:

- You need high throughput and low latency
- Your users are cost-sensitive
- You are building games, social apps, or high-frequency trading
- You are comfortable with Rust

Avoid Solana when:

- Maximum decentralization is required
- You need Ethereum ecosystem compatibility
- Your team only knows Solidity

SOLANA STRENGTHS

Speed: Sub-second finality, thousands of TPS

Cost: Fractions of a cent per transaction

Growing ecosystem: Strong in gaming, NFTs, DeFi

Institutional interest: ETF products launched in 2025

Developer experience: Anchor framework is mature

SOLANA WEAKNESSES

Centralization: High validator requirements

Outages: Network has experienced multiple outages historically

Learning curve: Rust is harder than Solidity

Smaller ecosystem: Fewer tools and libraries than Ethereum

SECTION 4: BITCOIN

WHAT IS BITCOIN

Bitcoin is the original blockchain. Created by Satoshi Nakamoto in 2009, Bitcoin solved the double-spend problem and created digital scarcity.

Bitcoin is not a smart contract platform. It is programmable money with limited scripting capabilities.

BITCOIN'S DESIGN PHILOSOPHY

Bitcoin maximizes:

- Decentralization: Anyone can run a full node on a Raspberry Pi
- Security: Proof of Work with massive hashrate
- Immutability: Hardest money to change

Bitcoin sacrifices:

- Speed: 7 transactions per second
- Programmability: Limited scripting
- Flexibility: Intentionally hard to upgrade

THE UTXO MODEL

Bitcoin uses UTXOs (Unspent Transaction Outputs) instead of accounts.

When you receive Bitcoin, you receive a UTXO -- a specific output that you can spend. When you spend, you consume UTXOs as inputs and create new UTXOs as outputs.

Example:

- Alice receives 1 BTC (creates UTXO-A worth 1 BTC)
- Alice receives 0.5 BTC (creates UTXO-B worth 0.5 BTC)
- Alice sends 1.2 BTC to Bob
 - Consumes UTXO-A and UTXO-B (1.5 BTC total)
 - Creates UTXO for Bob (1.2 BTC)
 - Creates change UTXO for Alice (0.3 BTC minus fee)

BITCOIN SCRIPT

Bitcoin has a simple scripting language. Most transactions use standard templates:

P2PKH (Pay to Public Key Hash):

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY  
OP_CHECKSIG
```

P2SH (Pay to Script Hash):

```
OP_HASH160 <scriptHash> OP_EQUAL
```

P2WPKH (Pay to Witness Public Key Hash):

```
OP_0 <pubKeyHash>
```

These scripts are not Turing-complete. They cannot loop. They cannot access external data. This is intentional.

LAYER 2: LIGHTNING NETWORK

The Lightning Network enables fast, cheap Bitcoin payments off-chain.

How it works:

1. Two parties open a channel by locking Bitcoin in a 2-of-2 multisig
2. They exchange signed transactions off-chain
3. Either party can close the channel and settle on-chain

Payments can route through multiple channels, enabling anyone to pay anyone without a direct channel.

CONNECTING TO BITCOIN

Here is how to connect to a Bitcoin node using Python:

```
from bitcoinrpc.authproxy import AuthServiceProxy

rpc_user = "your_rpc_user"
rpc_password = "your_rpc_password"
rpc_connection = AuthServiceProxy(f'http://{rpc_user}:{rpc_password}@127.0.0.1:8332')

block_count = rpc_connection.getblockcount()
print(f'Block height: {block_count}')

best_block_hash = rpc_connection.getbestblockhash()
print(f'Best block: {best_block_hash}')

network_info = rpc_connection.getnetworkinfo()
print(f'Version: {network_info["subversion"]}')

mempool_info = rpc_connection.getmempoolinfo()
print(f'Mempool size: {mempool_info["size"]} transactions')
```

ORDINALS AND INSCRIPTIONS

In 2023, Ordinals introduced a way to attach data to individual satoshis (the smallest Bitcoin unit). This enabled NFTs and tokens on Bitcoin.

Inscriptions are stored in transaction witness data. They can contain images, text, or other content up to 4MB.

This was controversial. Some see it as spam. Others see it as expanding Bitcoin's utility.

WHEN TO USE BITCOIN

Use Bitcoin when:

- You need the most secure, decentralized network
- You are building payment infrastructure
- You want censorship-resistant money
- You need long-term value storage

Avoid Bitcoin when:

- You need smart contracts
- You need high throughput
- You are building complex applications

BITCOIN STRENGTHS

Security: Most hashrate, most decentralized

Longevity: Running since 2009 without major issues

Simplicity: Does one thing extremely well

Network effect: Most recognized, most liquid

BITCOIN WEAKNESSES

Limited programmability: No smart contracts

Slow: 10-minute blocks, 7 TPS

Energy intensive: Proof of Work
Hard to build on: Limited developer tools

SECTION 5: POLYGON

WHAT IS POLYGON

Polygon started as Matic Network, an Ethereum sidechain. It evolved into a multi-chain scaling solution and is now focusing on ZK technology with Polygon 2.0.

POLYGON POS (PROOF OF STAKE)

The original Polygon is an EVM-compatible sidechain:

- Block time: 2 seconds
- Transaction cost: Fractions of a cent
- Validators: About 100 staked validators
- Security: Checkpoints posted to Ethereum

Polygon PoS is fast and cheap but less secure than Ethereum. It is a separate chain with its own validators, not a rollup that inherits Ethereum security.

POLYGON 2.0 AND AGGLAYER

Polygon 2.0 is a major restructuring:

Polygon zkEVM: A ZK rollup with full EVM compatibility. Transactions are proven with zero-knowledge proofs and settled on Ethereum.

AggLayer: An aggregation layer connecting all Polygon chains. Aims to make multiple chains feel like one.

POL Token: The new native token replacing MATIC, with expanded utility across the ecosystem.

THE COMPETITIVE CHALLENGE

Polygon faces increasing competition:

Arbitrum has overtaken Polygon in Total Value Locked (TVL).

Optimism excels with better Ethereum compatibility.

Base benefits from Coinbase's distribution and marketing.

Polygon's positioning as a multi-chain hub risks losing focus while competitors concentrate on pure Ethereum scaling.

CONNECTING TO POLYGON

The connection is identical to Ethereum (EVM compatible):

```
from web3 import Web3
```

```
polygon_rpc = "https://polygon-rpc.com"
```

```
w3 = Web3(Web3.HTTPProvider(polygon_rpc))
```

```
print(f'Connected: {w3.is_connected()}')
```

```
print(f'Chain ID: {w3.eth.chain_id}')
```

```
print(f'Block number: {w3.eth.block_number}')
```

```
print(f'Gas price: {w3.eth.gas_price / 10**9} gwei')
```

WHEN TO USE POLYGON

Use Polygon when:

- You need EVM compatibility with lower costs

- You have existing Solidity code
- Your users are on Polygon already
- You need a mature ecosystem with bridges and tools

Consider alternatives when:

- You need maximum Ethereum security (use Arbitrum or zkSync)
- You need maximum performance (use Solana)

POLYGON STRENGTHS

EVM compatible: All Ethereum tools work

Mature ecosystem: Many DApps, bridges, tools

Low costs: Very cheap transactions

Enterprise adoption: Strong partnerships

POLYGON WEAKNESSES

Sidechain security: PoS chain is not as secure as rollups

Competition: Arbitrum, Base, Optimism taking market share

Fragmented strategy: Multiple chains, unclear focus

Stagnating growth: User and developer adoption has plateaued

SECTION 6: OTHER NOTABLE PLATFORMS

ARBITRUM

The leading Ethereum L2 by TVL. Optimistic rollup with:

- Nitro upgrade for better performance
- Stylus for Rust/C++ smart contracts
- Strong DeFi ecosystem
- 7-day withdrawal period

OPTIMISM

Optimistic rollup powering the "Superchain" vision:

- Base is built on OP Stack
- Revenue sharing with OP token
- Retroactive public goods funding
- Growing ecosystem

BASE

Coinbase's L2 built on OP Stack:

- Coinbase user onboarding
- Strong retail focus
- Fast-growing ecosystem
- 13.94% of blockchain mindshare in 2025

ZKSYNC ERA

ZK rollup with native account abstraction:

- Smart contract wallets by default
- ZK proofs for security
- Growing DeFi ecosystem
- Faster withdrawals than optimistic rollups

BNB CHAIN (BINANCE SMART CHAIN)

Binance's EVM-compatible chain:

- Centralized (21 validators)
- Very low fees
- Large user base
- PancakeSwap ecosystem

SUI

Move-based blockchain from former Meta engineers:

- Object-centric model
- Parallel execution
- Growing gaming ecosystem
- 11.77% mindshare in 2025

AVALANCHE

Multi-chain platform:

- Subnets for custom chains
- EVM compatible C-Chain
- Sub-second finality
- Enterprise focus

SECTION 7: COMPARISON TABLE

PERFORMANCE COMPARISON

	Ethereum	Solana	Bitcoin	Polygon	PoS	Arbitrum
TPS	15-30	2,000+	7	65	40,000	
Block Time	12 sec	400 ms	10 min	2 sec	250 ms	
Finality	12 min	400 ms	60 min	2 sec	~1 week*	
Avg Tx Cost	\$2-20	\$0.00025	\$1-10	\$0.01	\$0.10	

* Arbitrum finality is immediate for most purposes but withdrawal to L1 takes 7 days

DECENTRALIZATION COMPARISON

	Ethereum	Solana	Bitcoin	Polygon	PoS	Arbitrum
Validators	900,000+	1,900	15,000+	100	Sequencer	

Node Req.	Low	High	Low	Medium	Low
Client Div.	Multiple	2 (soon 3)	Multiple	2	1

ECOSYSTEM COMPARISON

Ethereum	Solana	Bitcoin	Polygon	PoS	Arbitrum
TVL	\$50B+	\$5B+	N/A	\$1B+	\$3B+
DeFi	Dominant	Growing	Minimal	Mature	Strong
NFTs	Active	Active	Ordinals	Active	Limited
Gaming	Growing	Strong	None	Strong	Limited
Stablecoins	Dominant	Growing	Limited	Mature	Growing

DEVELOPMENT COMPARISON

Ethereum	Solana	Bitcoin	Polygon	PoS	Arbitrum
Language	Solidity	Rust	Script	Solidity	Solidity
Tooling	Excellent	Good	Limited	Excellent	Excellent
Docs	Excellent	Good	Good	Good	Good
Learning	Easy	Medium	Hard	Easy	Easy

SECTION 8: CHOOSING YOUR PLATFORM

DECISION FRAMEWORK

Answer these questions:

1. What type of application?

DeFi with large TVL -> Ethereum or Ethereum L2

Consumer app with many users -> Solana or Base

Gaming -> Solana or Polygon

Enterprise -> Polygon or Avalanche

Store of value -> Bitcoin

2. What is your transaction profile?

High value, infrequent -> Ethereum mainnet

High frequency, low value -> Solana or L2

Micropayments -> Solana or Lightning

3. What does your team know?

Solidity -> Ethereum, Polygon, Arbitrum, Base

Rust -> Solana

JavaScript only -> EVM chains with Hardhat/Foundry

4. Where are your users?

Already on Solana -> Solana

Already on Ethereum -> Ethereum or L2

New to crypto -> Low-fee chain with good UX

5. What is your timeline?

Launching fast -> Use what you know

Building for years -> Choose based on long-term vision

MULTI-CHAIN STRATEGY

Many successful projects deploy on multiple chains:

1. Start on one chain
2. Prove product-market fit
3. Expand to other chains
4. Use bridges for interoperability

This works well for:

- DeFi protocols (Uniswap, Aave on many chains)
- NFT collections (multi-chain minting)
- Games (different chains for different features)

THE PRACTICAL RECOMMENDATION

If you are new to blockchain development:

Start with an Ethereum L2 (Arbitrum or Base).

- Solidity is the easiest language
- Best tooling and documentation
- Low costs for experimentation
- Ethereum security guarantees
- Easy path to mainnet if needed

Once comfortable, expand to:

- Solana for performance-critical applications
- Other L2s for specific ecosystems
- Bitcoin for payment infrastructure

SECTION 9: CONNECTING TO MULTIPLE CHAINS

MULTI-CHAIN PYTHON SETUP

The following code connects to multiple chains:

```
from web3 import Web3
from solana.rpc.api import Client as SolanaClient

chains = {
    "ethereum": {
        "rpc": "https://mainnet.infura.io/v3/YOUR_KEY",
        "chain_id": 1,
```

```

"native": "ETH"
},
"polygon": {
"rpc": "https://polygon-rpc.com",
"chain_id": 137,
"native": "MATIC"
},
"arbitrum": {
"rpc": "https://arb1.arbitrum.io/rpc",
"chain_id": 42161,
"native": "ETH"
},
"base": {
"rpc": "https://mainnet.base.org",
"chain_id": 8453,
"native": "ETH"
},
"bsc": {
"rpc": "https://bsc-dataseed.binance.org",
"chain_id": 56,
"native": "BNB"
}
}

```

```

def connect_evm_chains():
    connections = {}
    for name, config in chains.items():
        w3 = Web3(Web3.HTTPProvider(config["rpc"]))
        connections[name] = {
            "web3": w3,
            "connected": w3.is_connected(),
            "block": w3.eth.block_number if w3.is_connected() else None
        }
    print(f'{name}: connected = {connections[name]
    ['connected']}, block = {connections[name]['block']}')
    return connections

```

```
def connect_solana():
    client = SolanaClient("https://api.mainnet-beta.solana.com")
    response = client.get_slot()
    print(f'solana: connected = True, slot = {response.value}')
    return client

def get_gas_prices(connections):
    print("\nGAS PRICES")
    print("-" * 40)
    for name, conn in connections.items():
        if conn["connected"]:
            gas_price = conn["web3"].eth.gas_price
            gwei = gas_price / 10**9
            print(f'{name}: {gwei:.2f} gwei')

evm_connections = connect_evm_chains()
solana_client = connect_solana()
get_gas_prices(evm_connections)
```

SECTION 10: WHAT YOU LEARNED

After reading this chapter, you understand:

1. The blockchain landscape: Market share, usage patterns, ecosystem differences
2. Ethereum: EVM architecture, Layer 2 strategy, when to use mainnet vs L2
3. Solana: Proof of History, high performance, Rust-based development, Firedancer
4. Bitcoin: UTXO model, limited scripting, Lightning Network, Ordinals

5. Polygon: Sidechain architecture, Polygon 2.0, competitive challenges

6. Other platforms: Arbitrum, Base, Optimism, zkSync, BNB Chain, Sui

7. Comparison: Performance, decentralization, ecosystem, development

8. Decision framework: How to choose the right platform for your project

You have code examples for connecting to:

- Ethereum and EVM-compatible chains
- Solana
- Bitcoin nodes
- Multiple chains simultaneously

The blockchain you choose is the foundation of everything you build. Choose based on your requirements, not hype. Start with what you know. Expand as you grow.

In the next chapter, we dive into smart contract development with Solidity: the language that powers Ethereum and all EVM-compatible chains.

CHAPTER 5

SMART CONTRACT DEVELOPMENT - SOLIDITY

A smart contract is code that runs on the blockchain. Once deployed, it cannot be changed. It executes exactly as written, every time, for everyone. No human intervention. No exceptions.

Solidity is the language that makes this possible on Ethereum and all EVM-compatible chains. This chapter teaches you

Solidity from the ground up.

SECTION 1: WHAT IS SOLIDITY

THE LANGUAGE

Solidity is a statically-typed, contract-oriented programming language. It was created specifically for writing smart contracts on Ethereum.

Key characteristics:

- Syntax similar to JavaScript and C + +
- Compiles to EVM bytecode
- Statically typed (types checked at compile time)
- Supports inheritance and libraries
- Has built-in primitives for blockchain interaction

THE DEVELOPMENT ENVIRONMENT

You need:

1. A code editor (VS Code with Solidity extension)
2. A compiler (solc, or use Hardhat/Foundry)
3. A local blockchain for testing (Hardhat Network, Anvil, or Ganache)
4. A wallet for deployment (private key management)

The standard setup in 2025:

Hardhat: JavaScript/TypeScript-based development framework

Foundry: Rust-based, faster, uses Solidity for tests

Remix: Browser-based IDE for quick prototyping

YOUR FIRST CONTRACT

Here is the simplest possible Solidity contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract HelloWorld {
    string public greeting = "Hello, World!";
}
```

Let us break this down:

The first line is the license identifier. SPDX-License-Identifier specifies the open source license. This is required by the compiler.

The pragma statement specifies the compiler version. The caret (^) means "this version or higher, but not the next major version." So ^0.8.20 accepts 0.8.20, 0.8.21, etc., but not 0.9.0.

The contract keyword defines a contract, similar to a class in other languages.

The string public greeting declares a state variable. It is stored on the blockchain permanently. The public keyword automatically creates a getter function.

SECTION 2: DATA TYPES

VALUE TYPES

Value types are passed by value. When you assign them, a

copy is made.

BOOLEANS

```
bool isActive = true;  
bool isComplete = false;
```

INTEGERS

Signed integers (can be negative):

int8 ranges from -128 to 127

int16 ranges from -32768 to 32767

int256 (or just int) is the default, ranging from -2^{255} to $2^{255}-1$

Unsigned integers (only positive):

uint8 ranges from 0 to 255

uint16 ranges from 0 to 65535

uint256 (or just uint) ranges from 0 to $2^{256}-1$

```
uint256 balance = 100000000000000000000;
```

```
int256 temperature = -20;
```

```
uint8 percentage = 100;
```

ADDRESS

The address type holds a 20-byte Ethereum address.

```
address wallet =
```

```
0x742d35Cc6634C0532925a3b844Bc9e7595f8fE21;
```

```
address payable recipient =
```

```
payable(0x742d35Cc6634C0532925a3b844Bc9e7595f8fE21);
```

The payable modifier allows the address to receive Ether.

BYTES

Fixed-size byte arrays:

```
bytes1 singleByte = 0xff;  
bytes32 hash = keccak256("hello");
```

Dynamic byte array:

```
bytes dynamicBytes = "hello world";
```

REFERENCE TYPES

Reference types point to data. Changes affect the original.

ARRAYS

Fixed-size arrays:

```
uint256[5] fixedArray;  
fixedArray[0] = 100;
```

Dynamic arrays:

```
uint256[] dynamicArray;  
dynamicArray.push(100);  
dynamicArray.push(200);  
uint256 length = dynamicArray.length;  
dynamicArray.pop();
```

STRINGS

```
string name = "Ethereum";  
string description = "A decentralized platform";
```

Strings are dynamic byte arrays with UTF-8 encoding. They cannot be indexed directly in Solidity. For character access, convert to bytes.

MAPPINGS

Mappings are hash tables. They map keys to values.

```
mapping(address => uint256) public balances;  
mapping(address => mapping(address => uint256)) public  
allowances;  
mapping(uint256 => string) public names;
```

Mappings cannot be iterated. You cannot get a list of keys. If you need iteration, maintain a separate array of keys.

STRUCTS

Structs group related data:

```
struct User {  
    address wallet;  
    string name;  
    uint256 balance;  
    bool isActive;  
}
```

```
User public admin;  
mapping(address => User) public users;
```

ENUMS

Enums define a type with a fixed set of values:

```
enum Status {  
    Pending,  
    Active,  
    Completed,  
    Cancelled  
}
```

```
Status public currentStatus = Status.Pending;
```

COMPLETE DATA TYPES EXAMPLE

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
contract DataTypes {  
    bool public isActive = true;
```

```
    uint256 public totalSupply = 1000000 * 10**18;  
    int256 public temperature = -10;
```

```
    address public owner;  
    address payable public treasury;
```

```
    bytes32 public documentHash;
```

```
    string public name = "MyToken";
```

```
    uint256[] public values;  
    address[3] public admins;
```

```
    mapping(address => uint256) public balances;
```

```
    struct Product {  
        uint256 id;  
        string name;  
        uint256 price;  
        bool available;
```

```
}
```

```
Product[] public products;
```

```
mapping(uint256 => Product) public productById;
```

```
enum OrderStatus { Created, Paid, Shipped, Delivered,  
Cancelled }
```

```
mapping(uint256 => OrderStatus) public orderStatuses;
```

```
constructor() {
```

```
owner = msg.sender;
```

```
treasury = payable(msg.sender);
```

```
values.push(100);
```

```
values.push(200);
```

```
admins[0] = msg.sender;
```

```
balances[msg.sender] = totalSupply;
```

```
products.push(Product(1, "Widget", 100, true));
```

```
productById[1] = products[0];
```

```
}
```

```
}
```

SECTION 3: FUNCTIONS

FUNCTION SYNTAX

```
function functionName(parameterType parameterName)
```

```
visibility mutability returns (returnType) {
```

```
// function body
```

```
}
```


VISIBILITY

public: Can be called externally and internally

external: Can only be called from outside the contract

internal: Can only be called from this contract or derived contracts

private: Can only be called from this contract

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.20;
```

```
contract Visibility {
```

```
    uint256 private secretValue = 42;
```

```
    uint256 internal sharedValue = 100;
```

```
    uint256 public openValue = 200;
```

```
    function publicFunction() public pure returns (string  
memory) {
```

```
        return "Anyone can call this";
```

```
    }
```

```
    function externalFunction() external pure returns (string  
memory) {
```

```
        return "Only external calls";
```

```
    }
```

```
    function internalFunction() internal pure returns (string  
memory) {
```

```
        return "This contract and children";
```

```
    }
```

```
    function privateFunction() private pure returns (string  
memory) {
```

```
        return "Only this contract";
```

```
    }
```

```
    function callInternal() public pure returns (string memory) {
```

```
return internalFunction();  
}
```

```
function callPrivate() public pure returns (string memory) {  
    return privateFunction();  
}  
}
```

STATE MUTABILITY

view: Reads state but does not modify it

pure: Does not read or modify state

payable: Can receive Ether

(no modifier): Can read and modify state

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
contract Mutability {  
    uint256 public value = 100;
```

```
    function getValue() public view returns (uint256) {  
        return value;  
    }
```

```
    function calculate(uint256 a, uint256 b) public pure returns  
(uint256) {  
        return a + b;  
    }
```

```
    function setValue(uint256 newValue) public {  
        value = newValue;  
    }
```

```
    function deposit() public payable {  
        // msg.value contains the Ether sent
```

```

}

function getBalance() public view returns (uint256) {
    return address(this).balance;
}
}

```

FUNCTION PARAMETERS AND RETURNS

Functions can have multiple parameters and return values:

```

function swap(uint256 a, uint256 b) public pure returns
(uint256, uint256) {
    return (b, a);
}

```

```

function getDetails() public pure returns (
    string memory name,
    uint256 age,
    bool active
) {
    return ("Alice", 30, true);
}

```

Named returns can be assigned directly:

```

function calculate(uint256 x) public pure returns (uint256
squared, uint256 cubed) {
    squared = x * x;
    cubed = x * x * x;
}

```

SPECIAL FUNCTIONS

Constructor runs once when the contract is deployed:

```
constructor(string memory _name, uint256 _value) {  
    name = _name;  
    value = _value;  
}
```

Receive function handles plain Ether transfers:

```
receive() external payable {  
    // Called when contract receives Ether with no data  
}
```

Fallback function handles calls that match no function:

```
fallback() external payable {  
    // Called when no function matches or data is sent with  
    Ether  
}
```

COMPLETE FUNCTIONS EXAMPLE

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
contract FunctionsExample {  
    address public owner;  
    uint256 public totalDeposits;  
    mapping(address => uint256) public deposits;
```

```
    event Deposited(address indexed user, uint256 amount);  
    event Withdrawn(address indexed user, uint256 amount);
```

```
    constructor() {  
        owner = msg.sender;  
    }
```

```
    function deposit() public payable {  
        require(msg.value > 0, "Must send ETH");
```

```
deposits[msg.sender] += msg.value;
totalDeposits += msg.value;
emit Deposited(msg.sender, msg.value);
}
```

```
function withdraw(uint256 amount) public {
    require(deposits[msg.sender] >= amount, "Insufficient
balance");
    deposits[msg.sender] -= amount;
    totalDeposits -= amount;
```

```
(bool success, ) = payable(msg.sender).call{value: amount}
("");
    require(success, "Transfer failed");
```

```
emit Withdrawn(msg.sender, amount);
}
```

```
function getMyBalance() public view returns (uint256) {
    return deposits[msg.sender];
}
```

```
function getContractBalance() public view returns (uint256) {
    return address(this).balance;
}
```

```
receive() external payable {
    deposits[msg.sender] += msg.value;
    totalDeposits += msg.value;
    emit Deposited(msg.sender, msg.value);
}
}
```

SECTION 4: MODIFIERS

WHAT ARE MODIFIERS

Modifiers are reusable code that wraps function execution. They check conditions before or after the function runs.

BASIC MODIFIER

```
modifier onlyOwner() {  
    require(msg.sender == owner, "Not owner");  
    _;  
}
```

```
function sensitiveAction() public onlyOwner {  
    // Only owner can call this  
}
```

The underscore (`_`) represents where the modified function's code runs.

MODIFIERS WITH PARAMETERS

```
modifier minimumAmount(uint256 amount) {  
    require(msg.value >= amount, "Below minimum");  
    _;  
}
```

```
function purchase() public payable minimumAmount(0.1  
ether) {  
    // Must send at least 0.1 ETH  
}
```

MULTIPLE MODIFIERS

Modifiers execute in order, left to right:

```

modifier notPaused() {
    require(!paused, "Contract paused");
    _;
}

```

```

modifier validAddress(address addr) {
    require(addr != address(0), "Invalid address");
    _;
}

```

```

function transfer(address to, uint256 amount)
    public
    notPaused
    validAddress(to)
{
    // Checks notPaused first, then validAddress
}

```

COMPLETE MODIFIERS EXAMPLE

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract ModifiersExample {
    address public owner;
    bool public paused;
    uint256 public minDeposit = 0.01 ether;
    mapping(address => bool) public admins;
    mapping(address => uint256) public lastAction;

    constructor() {
        owner = msg.sender;
        admins[msg.sender] = true;
    }

    modifier onlyOwner() {

```

```
require(msg.sender == owner, "Not owner");
└;
}
```

```
modifier onlyAdmin() {
require(admins[msg.sender], "Not admin");
└;
}
```

```
modifier notPaused() {
require(!paused, "Contract is paused");
└;
}
```

```
modifier validAddress(address addr) {
require(addr != address(0), "Zero address");
└;
}
```

```
modifier minValue(uint256 amount) {
require(msg.value >= amount, "Below minimum");
└;
}
```

```
modifier cooldown(uint256 duration) {
require(
block.timestamp >= lastAction[msg.sender] + duration,
"Cooldown active"
);
└;
lastAction[msg.sender] = block.timestamp;
}
```

```
function pause() public onlyOwner {
paused = true;
}
```



```
function unpause() public onlyOwner {
    paused = false;
}
```

```
function addAdmin(address admin) public onlyOwner
validAddress(admin) {
    admins[admin] = true;
}
```

```
function removeAdmin(address admin) public onlyOwner {
    admins[admin] = false;
}
```

```
function deposit() public payable notPaused
minValue(minDeposit) {
    // Deposit logic
}
```

```
function claim() public notPaused cooldown(1 hours) {
    // Can only claim once per hour
}
```

```
function adminAction() public onlyAdmin notPaused {
    // Only admins when not paused
}
}
```

SECTION 5: INHERITANCE

SINGLE INHERITANCE

Contracts can inherit from other contracts:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```

contract Animal {
    string public species;

    constructor(string memory _species) {
        species = _species;
    }

    function speak() public pure virtual returns (string memory) {
        return "...";
    }
}

contract Dog is Animal {
    constructor() Animal("Canine") {}

    function speak() public pure override returns (string memory)
    {
        return "Woof!";
    }
}

```

MULTIPLE INHERITANCE

Solidity supports multiple inheritance. Order matters -- list from most base-like to most derived:

```

contract A {
    function foo() public pure virtual returns (string memory) {
        return "A";
    }
}

contract B is A {
    function foo() public pure virtual override returns (string
memory) {

```

```
    return "B";  
  }  
}
```

```
contract C is A {  
  function foo() public pure virtual override returns (string  
memory) {  
    return "C";  
  }  
}
```

```
contract D is B, C {  
  function foo() public pure override(B, C) returns (string  
memory) {  
    return super.foo();  
  }  
}
```

ABSTRACT CONTRACTS

Abstract contracts cannot be deployed. They define interfaces that derived contracts must implement:

```
abstract contract Token {  
  function totalSupply() public view virtual returns (uint256);  
  function balanceOf(address account) public view virtual  
returns (uint256);  
  function transfer(address to, uint256 amount) public virtual  
returns (bool);  
}
```

```
contract MyToken is Token {  
  mapping(address => uint256) private _balances;  
  uint256 private _totalSupply;  
  
  function totalSupply() public view override returns (uint256)
```

```
{  
    return _totalSupply;  
}
```

```
function balanceOf(address account) public view override  
returns (uint256) {  
    return _balances[account];  
}
```

```
function transfer(address to, uint256 amount) public override  
returns (bool) {  
    _balances[msg.sender] -= amount;  
    _balances[to] += amount;  
    return true;  
}  
}
```

INTERFACES

Interfaces define function signatures without implementation. They cannot have state variables or constructors:

```
interface IERC20 {  
    function totalSupply() external view returns (uint256);  
    function balanceOf(address account) external view returns  
(uint256);  
    function transfer(address to, uint256 amount) external  
returns (bool);  
    function allowance(address owner, address spender) external  
view returns (uint256);  
    function approve(address spender, uint256 amount) external  
returns (bool);  
    function transferFrom(address from, address to, uint256  
amount) external returns (bool);
```

```
    event Transfer(address indexed from, address indexed to,
```

```

uint256 value);
    event Approval(address indexed owner, address indexed
spender, uint256 value);
}

```

USING INTERFACES TO INTERACT WITH OTHER CONTRACTS

```

interface IUniswapRouter {
    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
}

```

```

contract Trader {
    IUniswapRouter public router;

    constructor(address routerAddress) {
        router = IUniswapRouter(routerAddress);
    }

```

```

    function swap(
        address tokenIn,
        address tokenOut,
        uint256 amount
    ) external {
        address[] memory path = new address[](2);
        path[0] = tokenIn;
        path[1] = tokenOut;

        router.swapExactTokensForTokens(
            amount,

```

```

0,
path,
msg.sender,
block.timestamp + 300
);
}
}

```

COMPLETE INHERITANCE EXAMPLE

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

```

```

abstract contract Ownable {
    address public owner;

```

```

    event OwnershipTransferred(address indexed previousOwner,
address indexed newOwner);

```

```

    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

```

```

    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: not owner");
        _;
    }

```

```

    function transferOwnership(address newOwner) public
virtual onlyOwner {
    require(newOwner != address(0), "Ownable: zero address");
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
}
}

```

```
abstract contract Pausable is Ownable {  
    bool public paused;
```

```
    event Paused(address account);  
    event Unpaused(address account);
```

```
    modifier whenNotPaused() {  
        require(!paused, "Pausable: paused");  
        _;  
    }
```

```
    modifier whenPaused() {  
        require(paused, "Pausable: not paused");  
        _;  
    }
```

```
    function pause() public onlyOwner whenNotPaused {  
        paused = true;  
        emit Paused(msg.sender);  
    }
```

```
    function unpause() public onlyOwner whenPaused {  
        paused = false;  
        emit Unpaused(msg.sender);  
    }  
}
```

```
contract Vault is Pausable {  
    mapping(address => uint256) public balances;
```

```
    event Deposit(address indexed user, uint256 amount);  
    event Withdrawal(address indexed user, uint256 amount);
```

```
    function deposit() public payable whenNotPaused {  
        require(msg.value > 0, "Must send ETH");  
        balances[msg.sender] += msg.value;
```

```
emit Deposit(msg.sender, msg.value);
}
```

```
function withdraw(uint256 amount) public whenNotPaused {
  require(balances[msg.sender] >= amount, "Insufficient
  balance");
  balances[msg.sender] -= amount;
```

```
(bool success, ) = payable(msg.sender).call{value: amount}
("");
require(success, "Transfer failed");
```

```
emit Withdrawal(msg.sender, amount);
}
```

```
function emergencyWithdraw() public onlyOwner
whenPaused {
  uint256 balance = address(this).balance;
  (bool success, ) = payable(owner).call{value: balance}("");
  require(success, "Transfer failed");
}
}
```

SECTION 6: EVENTS AND LOGGING

WHAT ARE EVENTS

Events are logs stored on the blockchain. They are cheap to emit and cannot be read by contracts, but external applications can listen for them.

DECLARING EVENTS

event Transfer(address indexed from, address indexed to,


```
uint256 value);  
event Approval(address indexed owner, address indexed  
spender, uint256 value);  
event NewUser(address indexed user, string name, uint256  
timestamp);
```

The indexed keyword allows filtering by that parameter. You can have up to 3 indexed parameters.

EMITTING EVENTS

```
function transfer(address to, uint256 amount) public returns  
(bool) {  
    require(balances[msg.sender] >= amount, "Insufficient  
balance");  
  
    balances[msg.sender] -= amount;  
    balances[to] += amount;  
  
    emit Transfer(msg.sender, to, amount);  
  
    return true;  
}
```

LISTENING TO EVENTS (JAVASCRIPT/PYTHON)

Events are consumed off-chain. Here is how to listen in Python:

```
from web3 import Web3  
  
w3 = Web3(Web3.HTTPProvider("https://mainnet.infura.io/  
v3/YOUR_KEY"))  
  
contract_address = "0xContractAddress"  
contract_abi = [...]
```

```
contract = w3.eth.contract(address=contract_address,  
abi=contract_abi)
```

```
transfer_filter =  
contract.events.Transfer.create_filter(fromBlock='latest')
```

```
while True:  
    for event in transfer_filter.get_new_entries():  
        print(f"Transfer: {event['args']['from']} -> {event['args']  
        ['to']}: {event['args']['value']}")
```

COMPLETE EVENTS EXAMPLE

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
contract EventsExample {  
    mapping(address => uint256) public balances;  
    mapping(address => mapping(address => uint256))  
    public allowances;
```

```
    event Deposit(  
        address indexed user,  
        uint256 amount,  
        uint256 newBalance,  
        uint256 timestamp  
    );
```

```
    event Withdrawal(  
        address indexed user,  
        uint256 amount,  
        uint256 newBalance,  
        uint256 timestamp  
    );
```

```
event Transfer(  
    address indexed from,  
    address indexed to,  
    uint256 amount  
);
```

```
event Approval(  
    address indexed owner,  
    address indexed spender,  
    uint256 amount  
);
```

```
event Log(string message);  
event LogData(bytes data);
```

```
function deposit() public payable {  
    balances[msg.sender] += msg.value;
```

```
    emit Deposit(  
        msg.sender,  
        msg.value,  
        balances[msg.sender],  
        block.timestamp  
    );  
}
```

```
function withdraw(uint256 amount) public {  
    require(balances[msg.sender] >= amount, "Insufficient  
balance");
```

```
    balances[msg.sender] -= amount;
```

```
    (bool success, ) = payable(msg.sender).call{value: amount}  
    ("");  
    require(success, "Transfer failed");
```

```
    emit Withdrawal(  

```

```

msg.sender,
amount,
balances[msg.sender],
block.timestamp
);
}

```

```

function transfer(address to, uint256 amount) public returns
(bool) {
    require(balances[msg.sender] >= amount, "Insufficient
balance");
    require(to != address(0), "Invalid recipient");

    balances[msg.sender] -= amount;
    balances[to] += amount;

    emit Transfer(msg.sender, to, amount);

    return true;
}

```

```

function approve(address spender, uint256 amount) public
returns (bool) {
    allowances[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);

    return true;
}

```

```

function logMessage(string memory message) public {
    emit Log(message);
}
}

```

SECTION 7: ERROR HANDLING

REQUIRE, REVERT, ASSERT

Solidity has three ways to handle errors:

require: For input validation and preconditions. Refunds remaining gas.

```
function withdraw(uint256 amount) public {  
    require(amount > 0, "Amount must be positive");  
    require(balances[msg.sender] >= amount, "Insufficient  
balance");  
    // ...  
}
```

revert: For complex conditions. Also refunds remaining gas.

```
function process(uint256 value) public {  
    if (value < 100) {  
        revert("Value too low");  
    }  
    if (value > 1000) {  
        revert("Value too high");  
    }  
    // ...  
}
```

assert: For invariants that should never be false. Consumes all gas if triggered. Use only for internal errors.

```
function divide(uint256 a, uint256 b) internal pure returns  
(uint256) {  
    assert(b > 0);  
    return a / b;  
}
```

CUSTOM ERRORS

Custom errors are cheaper than string messages:

```
error Unauthorized(address caller, address required);
error InsufficientBalance(uint256 available, uint256 required);
error InvalidInput(string reason);
```

```
contract CustomErrors {
    address public owner;
    mapping(address => uint256) public balances;
```

```
    constructor() {
        owner = msg.sender;
    }
```

```
    function adminAction() public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender, owner);
        }
        // ...
    }
```

```
    function withdraw(uint256 amount) public {
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);
        }
        // ...
    }
}
```

TRY-CATCH

For handling errors from external calls:

```
interface IExternalContract {
```

```
function riskyFunction(uint256 value) external returns  
(uint256);  
}
```

```
contract TryCatchExample {  
    event Success(uint256 result);  
    event Failure(string reason);  
    event FailureData(bytes data);
```

```
    function callExternal(address target, uint256 value) public {  
        IExternalContract external_contract =  
        IExternalContract(target);
```

```
        try external_contract.riskyFunction(value) returns (uint256  
result) {  
            emit Success(result);  
        } catch Error(string memory reason) {  
            emit Failure(reason);  
        } catch (bytes memory data) {  
            emit FailureData(data);  
        }  
    }  
}
```

COMPLETE ERROR HANDLING EXAMPLE

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
error NotOwner(address caller);  
error NotAdmin(address caller);  
error InsufficientFunds(uint256 available, uint256 required);  
error InvalidAmount();  
error TransferFailed();  
error Paused();  
error NotPaused();
```

```
error ZeroAddress();
error DeadlineExpired(uint256 deadline, uint256 current);
```

```
contract ErrorHandling {
    address public owner;
    bool public paused;
    mapping(address => bool) public admins;
    mapping(address => uint256) public balances;
```

```
    constructor() {
        owner = msg.sender;
        admins[msg.sender] = true;
    }
```

```
    modifier onlyOwner() {
        if (msg.sender != owner) {
            revert NotOwner(msg.sender);
        }
        _;
    }
```

```
    modifier onlyAdmin() {
        if (!admins[msg.sender]) {
            revert NotAdmin(msg.sender);
        }
        _;
    }
```

```
    modifier whenNotPaused() {
        if (paused) {
            revert Paused();
        }
        _;
    }
```

```
    modifier validAddress(address addr) {
        if (addr == address(0)) {
```



```
revert ZeroAddress();  
}  
;  
}
```

```
modifier beforeDeadline(uint256 deadline) {  
    if (block.timestamp > deadline) {  
        revert DeadlineExpired(deadline, block.timestamp);  
    }  
    ;  
}
```

```
function pause() external onlyOwner {  
    if (paused) {  
        revert Paused();  
    }  
    paused = true;  
}
```

```
function unpause() external onlyOwner {  
    if (!paused) {  
        revert NotPaused();  
    }  
    paused = false;  
}
```

```
function deposit() external payable whenNotPaused {  
    if (msg.value == 0) {  
        revert InvalidAmount();  
    }  
    balances[msg.sender] += msg.value;  
}
```

```
function withdraw(uint256 amount) external whenNotPaused  
{  
    if (amount == 0) {  
        revert InvalidAmount();  
    }
```

```

}
if (balances[msg.sender] < amount) {
    revert InsufficientFunds(balances[msg.sender], amount);
}

balances[msg.sender] -= amount;

(bool success, ) = payable(msg.sender).call{value: amount}
("");
if (!success) {
    revert TransferFailed();
}
}

function timedAction(uint256 deadline) external
beforeDeadline(deadline) {
    // Action must complete before deadline
}

function addAdmin(address admin) external onlyOwner
validAddress(admin) {
    admins[admin] = true;
}
}

```

SECTION 8: PRACTICAL PATTERNS

CHECKS-EFFECTS-INTERACTIONS

Always follow this order to prevent reentrancy attacks:

1. Checks: Validate all inputs and conditions
2. Effects: Update state variables
3. Interactions: Call external contracts

```

function withdraw(uint256 amount) public {
    // CHECKS
    require(balances[msg.sender] >= amount, "Insufficient
balance");

    // EFFECTS
    balances[msg.sender] -= amount;

    // INTERACTIONS
    (bool success, ) = payable(msg.sender).call{value: amount}
("");
    require(success, "Transfer failed");
}

```

REENTRANCY GUARD

A modifier to prevent reentrancy:

```

contract ReentrancyGuard {
    uint256 private _status;
    uint256 private constant NOT_ENTERED = 1;
    uint256 private constant ENTERED = 2;

    constructor() {
        _status = NOT_ENTERED;
    }

    modifier nonReentrant() {
        require(_status != ENTERED, "ReentrancyGuard: reentrant
call");
        _status = ENTERED;
        _;
        _status = NOT_ENTERED;
    }
}

```

```

contract Vault is ReentrancyGuard {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) public nonReentrant {
        require(balances[msg.sender] >= amount);
        balances[msg.sender] -= amount;

        (bool success, ) = payable(msg.sender).call{value: amount}
        ("");
        require(success);
    }
}

```

PULL OVER PUSH

Instead of pushing payments to users, let them pull:

```

contract PullPayment {
    mapping(address => uint256) public pendingWithdrawals;

    function addFunds(address recipient, uint256 amount)
    internal {
        pendingWithdrawals[recipient] += amount;
    }

    function withdraw() public {
        uint256 amount = pendingWithdrawals[msg.sender];
        require(amount > 0, "Nothing to withdraw");

        pendingWithdrawals[msg.sender] = 0;

        (bool success, ) = payable(msg.sender).call{value: amount}
        ("");
        require(success, "Transfer failed");
    }
}

```

ACCESS CONTROL WITH ROLES

```
contract RoleBasedAccess {
    bytes32 public constant ADMIN_ROLE =
    keccak256("ADMIN_ROLE");
    bytes32 public constant MINTER_ROLE =
    keccak256("MINTER_ROLE");
    bytes32 public constant PAUSER_ROLE =
    keccak256("PAUSER_ROLE");

    mapping(bytes32 => mapping(address => bool)) private
    _roles;

    event RoleGranted(bytes32 indexed role, address indexed
    account);
    event RoleRevoked(bytes32 indexed role, address indexed
    account);

    constructor() {
        _roles[ADMIN_ROLE][msg.sender] = true;
        emit RoleGranted(ADMIN_ROLE, msg.sender);
    }

    modifier onlyRole(bytes32 role) {
        require(_roles[role][msg.sender], "Missing role");
        _;
    }

    function hasRole(bytes32 role, address account) public view
    returns (bool) {
        return _roles[role][account];
    }

    function grantRole(bytes32 role, address account) public
    onlyRole(ADMIN_ROLE) {
        _roles[role][account] = true;
```

```
emit RoleGranted(role, account);  
}
```

```
function revokeRole(bytes32 role, address account) public  
onlyRole(ADMIN_ROLE) {  
    _roles[role][account] = false;  
    emit RoleRevoked(role, account);  
}  
}
```

SECTION 9: COMPLETE CONTRACT EXAMPLE

Here is a complete, production-quality ERC20 token:

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;  
  
interface IERC20 {  
    function totalSupply() external view returns (uint256);  
    function balanceOf(address account) external view returns  
(uint256);  
    function transfer(address to, uint256 amount) external  
returns (bool);  
    function allowance(address owner, address spender) external  
view returns (uint256);  
    function approve(address spender, uint256 amount) external  
returns (bool);  
    function transferFrom(address from, address to, uint256  
amount) external returns (bool);  
  
    event Transfer(address indexed from, address indexed to,  
uint256 value);  
    event Approval(address indexed owner, address indexed  
spender, uint256 value);  
}
```

```
error ERC20InsufficientBalance(address sender, uint256
balance, uint256 needed);
error ERC20InvalidSender(address sender);
error ERC20InvalidReceiver(address receiver);
error ERC20InsufficientAllowance(address spender, uint256
allowance, uint256 needed);
error ERC20InvalidApprover(address approver);
error ERC20InvalidSpender(address spender);
```

```
contract ERC20Token is IERC20 {
```

```
    string public name;
    string public symbol;
    uint8 public decimals;
```

```
    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256))
private _allowances;
```

```
    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals,
        uint256 initialSupply
    ) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
```

```
        _mint(msg.sender, initialSupply * 10 ** _decimals);
    }
```

```
    function totalSupply() public view override returns (uint256)
    {
        return _totalSupply;
    }
```

```
function balanceOf(address account) public view override
returns (uint256) {
    return _balances[account];
}
```

```
function transfer(address to, uint256 amount) public override
returns (bool) {
    _transfer(msg.sender, to, amount);
    return true;
}
```

```
function allowance(address owner, address spender) public
view override returns (uint256) {
    return _allowances[owner][spender];
}
```

```
function approve(address spender, uint256 amount) public
override returns (bool) {
    _approve(msg.sender, spender, amount);
    return true;
}
```

```
function transferFrom(address from, address to, uint256
amount) public override returns (bool) {
    _spendAllowance(from, msg.sender, amount);
    _transfer(from, to, amount);
    return true;
}
```

```
function _transfer(address from, address to, uint256 amount)
internal {
    if (from == address(0)) {
        revert ERC20InvalidSender(address(0));
    }
    if (to == address(0)) {
        revert ERC20InvalidReceiver(address(0));
    }
}
```



```
}
```

```
uint256 fromBalance = _balances[from];  
if (fromBalance < amount) {  
    revert ERC20InsufficientBalance(from, fromBalance,  
amount);  
}
```

```
unchecked {  
    _balances[from] = fromBalance - amount;  
    _balances[to] += amount;  
}
```

```
emit Transfer(from, to, amount);  
}
```

```
function _approve(address owner, address spender, uint256  
amount) internal {  
    if (owner == address(0)) {  
        revert ERC20InvalidApprover(address(0));  
    }  
    if (spender == address(0)) {  
        revert ERC20InvalidSpender(address(0));  
    }
```

```
_allowances[owner][spender] = amount;  
emit Approval(owner, spender, amount);  
}
```

```
function _spendAllowance(address owner, address spender,  
uint256 amount) internal {  
    uint256 currentAllowance = _allowances[owner][spender];  
    if (currentAllowance != type(uint256).max) {  
        if (currentAllowance < amount) {  
            revert ERC20InsufficientAllowance(spender,  
currentAllowance, amount);  
        }
```

```
unchecked {  
    _allowances[owner][spender] = currentAllowance - amount;  
}  
}  
}
```

```
function _mint(address account, uint256 amount) internal {  
    if (account == address(0)) {  
        revert ERC20InvalidReceiver(address(0));  
    }
```

```
    _totalSupply += amount;  
    unchecked {  
        _balances[account] += amount;  
    }
```

```
    emit Transfer(address(0), account, amount);  
}
```

```
function _burn(address account, uint256 amount) internal {  
    if (account == address(0)) {  
        revert ERC20InvalidSender(address(0));  
    }
```

```
    uint256 accountBalance = _balances[account];  
    if (accountBalance < amount) {  
        revert ERC20InsufficientBalance(account, accountBalance,  
amount);  
    }
```

```
    unchecked {  
        _balances[account] = accountBalance - amount;  
        _totalSupply -= amount;  
    }
```

```
    emit Transfer(account, address(0), amount);  
}
```

}

SECTION 10: WHAT YOU LEARNED

After reading this chapter, you understand:

1. Solidity basics: pragma, contracts, state variables
2. Data types: booleans, integers, addresses, arrays, mappings, structs, enums
3. Functions: visibility, mutability, parameters, returns, special functions
4. Modifiers: creating reusable validation logic
5. Inheritance: single and multiple inheritance, abstract contracts, interfaces
6. Events: declaring, emitting, and listening to events
7. Error handling: require, revert, assert, custom errors, try-catch
8. Patterns: checks-effects-interactions, reentrancy guard, pull payments, role-based access

You have complete code examples for:

- Basic data type usage
- Function declarations and visibility
- Modifier creation and usage
- Contract inheritance hierarchies
- Event emission and handling
- Custom error definitions
- A complete ERC20 token implementation

Solidity is the foundation of Ethereum development. Master these concepts and you can build anything on EVM-compatible chains.

In the next chapter, we explore smart contract development with Rust for Solana, using the Anchor framework.

CHAPTER 6

SMART CONTRACT DEVELOPMENT - RUST FOR SOLANA

Solana programs are written in Rust. This is a different paradigm from Solidity. Rust is a systems programming language focused on safety and performance. Solana's architecture requires different mental models than Ethereum.

This chapter teaches you to build Solana programs using the Anchor framework.

SECTION 1: WHY RUST FOR SOLANA

THE LANGUAGE CHOICE

Solana chose Rust for several reasons:

Performance: Rust compiles to native machine code. No virtual machine overhead. Programs execute at near-hardware speeds.

Memory safety: Rust's ownership system prevents common bugs like null pointer dereferences, buffer overflows, and data races at compile time.

No garbage collection: Rust manages memory without a garbage collector. This enables predictable performance --

critical for blockchain where every microsecond matters.

Growing ecosystem: Rust has excellent tooling, package management (Cargo), and a helpful compiler that explains errors clearly.

RUST VS SOLIDITY

Solidity is designed for blockchain. Rust is a general-purpose systems language adapted for blockchain.

Solidity Rust

Learning Curve Moderate Steep

Memory Management Automatic Manual (ownership)

Type System Static, simple Static, complex

Error Handling require/revert Result/Option types

Execution EVM interpreted Native compiled

Performance Slower Much faster

THE ANCHOR FRAMEWORK

Writing raw Solana programs in Rust is complex. Anchor abstracts much of this complexity:

- Automatic account serialization and deserialization
- Declarative account validation
- Simplified error handling
- Built-in testing framework
- IDL generation for client integration

Anchor is to Solana what Hardhat is to Ethereum -- the standard development framework.

SECTION 2: SOLANA'S PROGRAMMING MODEL

ACCOUNTS, NOT STORAGE

Ethereum contracts have internal storage. Solana programs are stateless. All data lives in accounts.

An account has:

- Address (public key)
- Owner (program that controls it)
- Lamports (balance in smallest SOL unit)
- Data (arbitrary bytes)
- Executable flag (is this a program?)

Programs read and write to accounts they are passed. They cannot access accounts not explicitly provided in the transaction.

PROGRAMS VS SMART CONTRACTS

In Ethereum, a smart contract is both code and data. In Solana, programs (code) and accounts (data) are separate.

This separation enables:

- Programs can be upgraded without migrating data
- Multiple programs can operate on shared data
- Parallel transaction execution (non-overlapping accounts)

THE TRANSACTION MODEL

A Solana transaction contains:

- A list of instructions
- Each instruction specifies:
 - Program to call
 - Accounts to use
 - Instruction data (parameters)

Multiple instructions can execute atomically in one transaction.

PROGRAM DERIVED ADDRESSES (PDAs)

PDAs are deterministic addresses derived from seeds and a program ID. They have no private key -- only the program can sign for them.

PDAs enable:

- Predictable addresses for program-owned accounts
- Cross-program invocation with program authority
- Storing data at known locations

The derivation:

```
seeds = ["user", user_pubkey]
pda = find_program_address(seeds, program_id)
```

SECTION 3: SETTING UP THE ENVIRONMENT

INSTALL RUST

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

Verify installation:

```
rustc --version
cargo --version
```

INSTALL SOLANA CLI

```
sh -c "$(curl -sSfL https://release.solana.com/stable/install)"
export PATH="$HOME/.local/share/solana/install/
active_release/bin:$PATH"
```

Verify and configure:

```
solana --version
solana config set --url devnet
solana-keygen new
```

INSTALL ANCHOR

```
cargo install --git https://github.com/coral-xyz/anchor avm --
locked
avm install latest
avm use latest
```

Verify:

```
anchor --version
```

CREATE A NEW PROJECT

```
anchor init my_project
cd my_project
```

This creates:

```
my_project/
  programs/
  my_project/
  src/
  lib.rs # Main program code
  Cargo.toml # Rust dependencies
  tests/
```



```
my_project.ts # TypeScript tests
Anchor.toml # Anchor configuration
Cargo.toml # Workspace configuration
package.json # Node dependencies
```

SECTION 4: YOUR FIRST ANCHOR PROGRAM

THE BASIC STRUCTURE

Every Anchor program follows this structure:

```
use anchor_lang::prelude::*;

declare_id!
("YourProgramIdHere11111111111111111111111111111111");

#[program]
pub mod my_program {
    use super::*;

    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize {}
```

Breaking this down:

`use anchor_lang::prelude::*` imports all common Anchor types.

`declare_id!` sets the program's address. This is verified at runtime.

The `#[program]` macro marks the module containing instruction handlers.

Each function in the module is an instruction. It takes a Context and returns Result<()>.

The `#[derive(Accounts)]` struct defines what accounts the instruction needs.

A COUNTER PROGRAM

Let us build a program that stores and increments a counter:

```
use anchor_lang::prelude::*;
```

```
declare_id!  
("Counter1111111111111111111111111111111111111111");
```

```
#[program]
pub mod counter {
  use super::*;
```

```
pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
    let counter = &mut ctx.accounts.counter;
    counter.count = 0;
    counter.authority = ctx.accounts.authority.key();
    msg!("Counter initialized with count: {}", counter.count);
    Ok(())
}
```

```
pub fn increment(ctx: Context<Increment>) ->
Result<()> {
  let counter = &mut ctx.accounts.counter;
  counter.count += 1;
  msg!("Counter incremented to: {}", counter.count);
  Ok(())
}
```

```
}
```

```
pub fn decrement(ctx: Context<Decrement>) ->
Result<()> {
    let counter = &mut ctx.accounts.counter;
    require!(counter.count > 0, CounterError::UnderflowError);
    counter.count -= 1;
    msg!("Counter decremented to: {}", counter.count);
    Ok(())
}
```

```
pub fn set(ctx: Context<Set>, value: u64) -> Result<()> {
    let counter = &mut ctx.accounts.counter;
    counter.count = value;
    msg!("Counter set to: {}", counter.count);
    Ok(())
}
}
```

```
#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(
        init,
        payer = authority,
        space = 8 + Counter::INIT_SPACE
    )]
    pub counter: Account<'info, Counter>,
}
```

```
#[account(mut)]
pub authority: Signer<'info> ,
```

```
pub system_program: Program<'info, System> ,
}
```

```
#[derive(Accounts)]
pub struct Increment<'info> {
    #[account(mut)]
```

```
pub counter: Account<'info, Counter> ,  
}
```

```
#[derive(Accounts)]  
pub struct Decrement<'info> {  
    #[account(mut)]  
    pub counter: Account<'info, Counter> ,  
}
```

```
#[derive(Accounts)]  
pub struct Set<'info> {  
    #[account(  
        mut,  
        has_one = authority  
    )]  
    pub counter: Account<'info, Counter> ,
```

```
    pub authority: Signer<'info> ,  
}
```

```
#[account]  
#[derive(InitSpace)]  
pub struct Counter {  
    pub count: u64,  
    pub authority: Pubkey,  
}
```

```
#[error_code]  
pub enum CounterError {  
    #[msg("Cannot decrement below zero")]  
    UnderflowError,  
}
```

Let us examine each part in detail.

SECTION 5: ACCOUNT DEFINITIONS

THE ACCOUNT STRUCT

The Counter account stores our data:

```
#[account]
#[derive(InitSpace)]
pub struct Counter {
    pub count: u64,
    pub authority: Pubkey,
}
```

The `#[account]` macro:

- Adds an 8-byte discriminator (unique identifier)
- Implements serialization/deserialization
- Implements account validation

The `#[derive(InitSpace)]` macro calculates the space needed:

- `u64` = 8 bytes
- `Pubkey` = 32 bytes
- Total = 40 bytes (plus 8-byte discriminator = 48 bytes)

ACCOUNT CONSTRAINTS

The `#[account(...)]` attribute adds validation:

```
#[account(init, payer = authority, space = 8 +
Counter::INIT_SPACE)]
```

`init`: Create a new account

`payer`: Who pays for rent

`space`: How many bytes to allocate

```
#[account(mut)]
```

mut: This account will be modified

`#[account(mut, has_one = authority)]`

has_one: Verify that counter.authority equals the authority account passed in

COMMON CONSTRAINTS

`#[account(init, payer = payer, space = 8 + 100)]`
Creates a new account with 108 bytes of space

`#[account(mut)]`
Account will be modified

`#[account(has_one = owner)]`
Verifies `account.owner == owner` passed in transaction

`#[account(constraint = amount > 0)]`
Custom constraint with error

`#[account(seeds = [b"seed", user.key().as_ref()], bump)]`
PDA derivation

`#[account(close = destination)]`
Close account and send lamports to destination

ACCOUNT TYPES

Account <'info, T>: Deserialized account of type T

Signer <'info>: Must be a signer of the transaction

Program <'info, T>: A program account

SystemAccount <'info>: A system-owned account

UncheckedAccount <'info>: No validation (use carefully)

SECTION 6: PROGRAM DERIVED ADDRESSES

WHAT ARE PDAs

PDAs are addresses that:

- Are derived deterministically from seeds
- Have no corresponding private key
- Can only be signed for by the program

This enables programs to "own" accounts and sign transactions programmatically.

DERIVING A PDA

In the program:

```
#[derive(Accounts)]
pub struct CreateVault <'info> {
  #[account(
    init,
    seeds = [b"vault", user.key().as_ref()],
    bump,
    payer = user,
    space = 8 + Vault::INIT_SPACE
  )]
  pub vault: Account <'info, Vault>,

  #[account(mut)]
  pub user: Signer <'info>,

  pub system_program: Program <'info, System>,
}
```

The seeds define the derivation:

- b"vault" is a literal byte string
- user.key().as_ref() is the user's public key as bytes

The bump is a single byte that makes the address valid (off the ed25519 curve).

STORING THE BUMP

Save the bump for future lookups:

```
#[account]
#[derive(InitSpace)]
pub struct Vault {
    pub owner: Pubkey,
    pub balance: u64,
    pub bump: u8,
}

pub fn create_vault(ctx: Context<CreateVault>) ->
Result<()> {
    let vault = &mut ctx.accounts.vault;
    vault.owner = ctx.accounts.user.key();
    vault.balance = 0;
    vault.bump = ctx.bumps.vault;
    Ok(())
}
```

SIGNING WITH A PDA

When a program needs to sign a CPI (cross-program invocation):

```
pub fn withdraw(ctx: Context<Withdraw>, amount: u64) ->
Result<()> {
    let vault = &ctx.accounts.vault;
```



```

let vault = &mut ctx.accounts.vault;
vault.owner = ctx.accounts.owner.key();
vault.bump = ctx.bumps.vault;
Ok(())
}

```

```

pub fn deposit(ctx: Context<Deposit>, amount: u64) ->
Result<()> {

```

```

let cpi_context = CpiContext::new(
ctx.accounts.system_program.to_account_info(),
Transfer {
from: ctx.accounts.owner.to_account_info(),
to: ctx.accounts.vault.to_account_info(),
},
);

```

```

transfer(cpi_context, amount)?;

```

```

Ok(())
}

```

```

pub fn withdraw(ctx: Context<Withdraw>, amount: u64) -
> Result<()> {

```

```

let vault = &ctx.accounts.vault;

```

```

let seeds = &[
b"vault",
vault.owner.as_ref(),
&[vault.bump],
];

```

```

let signer_seeds = &[&seeds[..]];

```

```

let cpi_context = CpiContext::new_with_signer(
ctx.accounts.system_program.to_account_info(),
Transfer {
from: ctx.accounts.vault.to_account_info(),
to: ctx.accounts.owner.to_account_info(),
}

```

```
},  
signer_seeds,  
);
```

```
transfer(cpi_context, amount)?;
```

```
Ok()  
}  
}
```

```
#[derive(Accounts)]  
pub struct Initialize<'info> {  
    #[account(  
        init,  
        seeds = [b"vault", owner.key().as_ref()],  
        bump,  
        payer = owner,  
        space = 8 + Vault::INIT_SPACE  
    )]  
    pub vault: Account<'info, Vault> ,
```

```
    #[account(mut)]  
    pub owner: Signer<'info> ,
```

```
    pub system_program: Program<'info, System> ,  
}
```

```
#[derive(Accounts)]  
pub struct Deposit<'info> {  
    #[account(  
        mut,  
        seeds = [b"vault", owner.key().as_ref()],  
        bump = vault.bump,  
    )]  
    pub vault: Account<'info, Vault> ,
```

```
    #[account(mut)]
```

```

pub owner: Signer <'info> ,

pub system_program: Program <'info, System> ,
}

#[derive(Accounts)]
pub struct Withdraw <'info> {
    #[account(
        mut,
        seeds = [b"vault", owner.key().as_ref()],
        bump = vault.bump,
        has_one = owner,
    )]
    pub vault: Account <'info, Vault> ,

    #[account(mut)]
    pub owner: Signer <'info> ,

    pub system_program: Program <'info, System> ,
}

#[account]
#[derive(InitSpace)]
pub struct Vault {
    pub owner: Pubkey,
    pub bump: u8,
}

```

SECTION 7: CROSS-PROGRAM INVOCATION

WHAT IS CPI

CPI allows one program to call another. This is how composability works in Solana.

Examples:

- Your program calling the Token program to transfer tokens
- Your program calling the System program to create accounts
- Your program calling another custom program

BASIC CPI

To transfer SOL using the System program:

```
use anchor_lang::system_program::{transfer, Transfer};
```

```
pub fn pay(ctx: Context<Pay>, amount: u64) ->  
Result<()> {  
    let cpi_context = CpiContext::new(  
        ctx.accounts.system_program.to_account_info(),  
        Transfer {  
            from: ctx.accounts.payer.to_account_info(),  
            to: ctx.accounts.recipient.to_account_info(),  
        },  
    );  
  
    transfer(cpi_context, amount)?;  
  
    Ok(())  
}
```

```
#[derive(Accounts)]  
pub struct Pay<'info> {  
    #[account(mut)]  
    pub payer: Signer<'info>,  
  
    #[account(mut)]  
    pub recipient: SystemAccount<'info>,  
  
    pub system_program: Program<'info, System>,  
}
```

CPI WITH SIGNER SEEDS

When the PDA needs to sign:

```
let seeds = &[b"authority", &[bump]];
let signer_seeds = &[&seeds[..]];

let cpi_context = CpiContext::new_with_signer(
    program.to_account_info(),
    instruction_accounts,
    signer_seeds,
);
```

TOKEN TRANSFERS

To transfer SPL tokens:

```
use anchor_spl::token::{self, Token, TokenAccount, Transfer};

pub fn transfer_tokens(ctx: Context<TransferTokens>,
    amount: u64) -> Result<()> {
    let cpi_accounts = Transfer {
        from: ctx.accounts.from_token_account.to_account_info(),
        to: ctx.accounts.to_token_account.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
    };

    let cpi_program =
        ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

    token::transfer(cpi_ctx, amount)?;

    Ok(())
}
```

```
#[derive(Accounts)]
pub struct TransferTokens <'info> {
    #[account(mut)]
    pub from_token_account: Account <'info, TokenAccount>,

    #[account(mut)]
    pub to_token_account: Account <'info, TokenAccount>,

    pub authority: Signer <'info>,

    pub token_program: Program <'info, Token>,
}
```

SECTION 8: ERROR HANDLING

CUSTOM ERRORS

Define errors with the `#[error_code]` macro:

```
#[error_code]
pub enum GameError {
    #[msg("Game has already started")]
    GameAlreadyStarted,

    #[msg("Game has not started yet")]
    GameNotStarted,

    #[msg("Player has already joined")]
    PlayerAlreadyJoined,

    #[msg("Maximum players reached")]
    MaxPlayersReached,

    #[msg("Insufficient funds")]
    InsufficientFunds,
```

```
#[msg("Invalid move")]
InvalidMove,
}
```

USING ERRORS

With `require!` macro:

```
pub fn join_game(ctx: Context<JoinGame>) -> Result<()>
{
    let game = &mut ctx.accounts.game;

    require(!game.started, GameError::GameAlreadyStarted);
    require!(game.players.len() < 4,
GameError::MaxPlayersReached);

    game.players.push(ctx.accounts.player.key());

    Ok(())
}
```

With `if` statements:

```
pub fn make_move(ctx: Context<MakeMove>, position: u8) -
> Result<()> {
    let game = &mut ctx.accounts.game;

    if !game.started {
        return Err(GameError::GameNotStarted.into());
    }

    if position > 8 {
        return Err(GameError::InvalidMove.into());
    }

    Ok(())
}
```



```
}
```

REQUIRE MACROS

`require!(condition, Error)`: Basic condition check

`require_eq!(a, b, Error)`: Equality check

`require_neq!(a, b, Error)`: Inequality check

`require_gt!(a, b, Error)`: Greater than

`require_gte!(a, b, Error)`: Greater than or equal

`require_keys_eq!(a, b, Error)`: Pubkey equality

SECTION 9: TESTING ANCHOR PROGRAMS

TYPESCRIPT TESTS

Anchor generates TypeScript tests. Here is a test for our counter:

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { Counter } from "../target/types/counter";
import { expect } from "chai";

describe("counter", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.Counter as
  Program<Counter>;
  const counterKeypair = anchor.web3.Keypair.generate();

  it("Initializes the counter", async () => {
    await program.methods
      .initialize()
```

```
.accounts({
  counter: counterKeypair.publicKey,
  authority: provider.wallet.publicKey,
  systemProgram: anchor.web3.SystemProgram.programId,
})
.signers([counterKeypair])
.rpc();
```

```
const counter = await program.account.counter.fetch(
  counterKeypair.publicKey
);
```

```
expect(counter.count.toNumber()).to.equal(0);
expect(counter.authority.toString()).to.equal(
  provider.wallet.publicKey.toString()
);
});
```

```
it("Increments the counter", async () => {
  await program.methods
    .increment()
    .accounts({
      counter: counterKeypair.publicKey,
    })
    .rpc();
```

```
const counter = await program.account.counter.fetch(
  counterKeypair.publicKey
);
```

```
expect(counter.count.toNumber()).to.equal(1);
});
```

```
it("Increments multiple times", async () => {
  await program.methods.increment().accounts({
    counter: counterKeypair.publicKey,
  }).rpc();
```

```
await program.methods.increment().accounts({
  counter: counterKeypair.publicKey,
}).rpc();
```

```
const counter = await program.account.counter.fetch(
  counterKeypair.publicKey
);
```

```
expect(counter.count.toNumber()).to.equal(3);
});
```

```
it("Sets the counter", async () => {
  await program.methods
    .set(new anchor.BN(100))
    .accounts({
      counter: counterKeypair.publicKey,
      authority: provider.wallet.publicKey,
    })
    .rpc();
```

```
const counter = await program.account.counter.fetch(
  counterKeypair.publicKey
);
```

```
expect(counter.count.toNumber()).to.equal(100);
});
```

```
it("Decrements the counter", async () => {
  await program.methods
    .decrement()
    .accounts({
      counter: counterKeypair.publicKey,
    })
    .rpc();
```

```
const counter = await program.account.counter.fetch(
```

```
counterKeypair.publicKey  
);
```

```
expect(counter.count.toNumber()).to.equal(99);  
});
```

```
it("Fails to decrement below zero", async () => {  
  await program.methods  
    .set(new anchor.BN(0))  
    .accounts({  
      counter: counterKeypair.publicKey,  
      authority: provider.wallet.publicKey,  
    })  
    .rpc();
```

```
  try {  
    await program.methods  
      .decrement()  
      .accounts({  
        counter: counterKeypair.publicKey,  
      })  
      .rpc();
```

```
    expect.fail("Should have thrown an error");  
  } catch (error) {  
    expect(error.message).to.include("UnderflowError");  
  }  
});  
});
```

RUNNING TESTS

Build the program:

```
anchor build
```

Start a local validator and run tests:

`anchor test`

Run tests without restarting validator:

`anchor test --skip-local-validator`

SECTION 10: BUILDING AND DEPLOYING

BUILD THE PROGRAM

`anchor build`

This creates:

- `target/deploy/program_name.so` (the compiled program)
- `target/idl/program_name.json` (the Interface Definition Language)
- `target/types/program_name.ts` (TypeScript types)

GET THE PROGRAM ID

`solana address -k target/deploy/program_name-keypair.json`

Update the `declare_id!` in your program with this address.
Also update `Anchor.toml`:

```
[programs.devnet]
my_program = "YourProgramIdHere"
```

DEPLOY TO DEVNET

Ensure you have devnet SOL:

solana airdrop 2

Deploy:

anchor deploy

Or deploy to a specific network:

```
anchor deploy --provider.cluster devnet
```

VERIFY DEPLOYMENT

solana program show <PROGRAM_ID>

UPGRADE A PROGRAM

Programs can be upgraded if deployed with an upgrade authority:

```
anchor upgrade target/deploy/program_name.so --program-id
<PROGRAM_ID>
```

SECTION 11: COMPLETE TOKEN PROGRAM EXAMPLE

Here is a complete SPL-like token program:

```
use anchor_lang::prelude::*;
```

```
declare_id!  
("Token1111111111111111111111111111111111111111");
```

```
#[program]
pub mod simple_token {
```

```
use super::*;
```

```
pub fn initialize_mint(
    ctx: Context<InitializeMint>,
    decimals: u8,
) -> Result<()> {
    let mint = &mut ctx.accounts.mint;
    mint.authority = ctx.accounts.authority.key();
    mint.supply = 0;
    mint.decimals = decimals;
    Ok(())
}
```

```
pub fn initialize_account(ctx: Context<InitializeAccount>) -
> Result<()> {
    let token_account = &mut ctx.accounts.token_account;
    token_account.mint = ctx.accounts.mint.key();
    token_account.owner = ctx.accounts.owner.key();
    token_account.balance = 0;
    Ok(())
}
```

```
pub fn mint_tokens(ctx: Context<MintTokens>, amount:
u64) -> Result<()> {
    let mint = &mut ctx.accounts.mint;
    let token_account = &mut ctx.accounts.token_account;

    mint.supply = mint.supply.checked_add(amount)
        .ok_or(TokenError::Overflow)?;

    token_account.balance =
        token_account.balance.checked_add(amount)
            .ok_or(TokenError::Overflow)?;

    emit!(TokensMinted {
        mint: mint.key(),
        to: token_account.key(),
```

```
amount,  
});
```

```
Ok()  
}
```

```
pub fn transfer(ctx: Context<TransferTokens>, amount:  
u64) -> Result<()> {
```

```
    let from = &mut ctx.accounts.from;  
    let to = &mut ctx.accounts.to;
```

```
    require!(from.balance >= amount,  
TokenError::InsufficientBalance);  
    require!(from.mint == to.mint, TokenError::MintMismatch);
```

```
    from.balance = from.balance.checked_sub(amount)  
    .ok_or(TokenError::Underflow)?;
```

```
    to.balance = to.balance.checked_add(amount)  
    .ok_or(TokenError::Overflow)?;
```

```
    emit!(TokensTransferred {  
        from: from.key(),  
        to: to.key(),  
        amount,  
    });
```

```
Ok()  
}
```

```
pub fn burn(ctx: Context<BurnTokens>, amount: u64) ->  
Result<()> {
```

```
    let mint = &mut ctx.accounts.mint;  
    let token_account = &mut ctx.accounts.token_account;
```

```
    require!(token_account.balance >= amount,  
TokenError::InsufficientBalance);
```



```

token_account.balance =
token_account.balance.checked_sub(amount)
.ok_or(TokenError::Underflow)?;

mint.supply = mint.supply.checked_sub(amount)
.ok_or(TokenError::Underflow)?;

emit!(TokensBurned {
mint: mint.key(),
from: token_account.key(),
amount,
});

Ok(())
}
}

```

```

#[derive(Accounts)]
pub struct InitializeMint <'info> {
#[account(
init,
payer = authority,
space = 8 + Mint::INIT_SPACE
)]
pub mint: Account <'info, Mint>,

#[account(mut)]
pub authority: Signer <'info>,

pub system_program: Program <'info, System>,
}

```

```

#[derive(Accounts)]
pub struct InitializeAccount <'info> {
#[account(
init,

```

```

payer = payer,
space = 8 + TokenAccount::INIT_SPACE
)]
pub token_account: Account<'info, TokenAccount>,

pub mint: Account<'info, Mint>,
pub owner: SystemAccount<'info>,

#[account(mut)]
pub payer: Signer<'info>,

pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct MintTokens<'info> {
  #[account(mut, has_one = authority)]
  pub mint: Account<'info, Mint>,

  #[account(mut, constraint = token_account.mint ==
mint.key())]
  pub token_account: Account<'info, TokenAccount>,

  pub authority: Signer<'info>,
}

#[derive(Accounts)]
pub struct TransferTokens<'info> {
  #[account(mut, constraint = from.owner == owner.key())]
  pub from: Account<'info, TokenAccount>,

  #[account(mut)]
  pub to: Account<'info, TokenAccount>,

  pub owner: Signer<'info>,
}

```

```
#[derive(Accounts)]
pub struct BurnTokens <'info> {
    #[account(mut)]
    pub mint: Account <'info, Mint>,

    #[account(
        mut,
        constraint = token_account.mint == mint.key(),
        constraint = token_account.owner == owner.key()
    )]
    pub token_account: Account <'info, TokenAccount>,

    pub owner: Signer <'info>,
}
```

```
#[account]
#[derive(InitSpace)]
pub struct Mint {
    pub authority: Pubkey,
    pub supply: u64,
    pub decimals: u8,
}
```

```
#[account]
#[derive(InitSpace)]
pub struct TokenAccount {
    pub mint: Pubkey,
    pub owner: Pubkey,
    pub balance: u64,
}
```

```
#[error_code]
pub enum TokenError {
    #[msg("Insufficient balance for transfer")]
    InsufficientBalance,
    #[msg("Arithmetic overflow")]
    Overflow,
}
```

```
#[msg("Arithmetic underflow")]
Underflow,
#[msg("Token account mint does not match")]
MintMismatch,
}
```

```
#[event]
pub struct TokensMinted {
    pub mint: Pubkey,
    pub to: Pubkey,
    pub amount: u64,
}
```

```
#[event]
pub struct TokensTransferred {
    pub from: Pubkey,
    pub to: Pubkey,
    pub amount: u64,
}
```

```
#[event]
pub struct TokensBurned {
    pub mint: Pubkey,
    pub from: Pubkey,
    pub amount: u64,
}
```

SECTION 12: WHAT YOU LEARNED

After reading this chapter, you understand:

1. Why Solana uses Rust: Performance, safety, no garbage collection
2. Solana's model: Accounts vs storage, programs vs contracts,

PDAs

3. Anchor basics: Program structure, accounts, constraints
4. Account definitions: The `#[account]` macro, space calculation, constraints
5. PDAs: Derivation, storing bumps, signing with PDAs
6. CPI: Calling other programs, signer seeds, token transfers
7. Error handling: Custom errors, require macros, Result types
8. Testing: TypeScript tests, running with Anchor
9. Deployment: Building, deploying, upgrading programs

You have complete code examples for:

- A counter program with all CRUD operations
- A vault program with PDA-based ownership
- A token program with mint, transfer, and burn

Solana development requires a different mental model than Ethereum. Accounts are separate from programs. Data must be passed explicitly. But the performance benefits are enormous -- thousands of transactions per second at fractions of a cent.

In the next chapter, we explore contract patterns and token standards: ERC-20, ERC-721, ERC-1155, and their implementations.

CHAPTER 7

CONTRACT PATTERNS AND STANDARDS

Smart contracts do not exist in isolation. They interact with wallets, exchanges, other contracts, and user interfaces. For this ecosystem to work, contracts must speak the same

language.

Standards define that language. Patterns provide proven solutions to common problems.

This chapter covers the essential standards and patterns every blockchain developer must know.

SECTION 1: WHY STANDARDS MATTER

THE INTEROPERABILITY PROBLEM

Imagine every token contract had different function names:

- Contract A uses `sendTokens(address, uint256)`
- Contract B uses `transfer(address, uint256)`
- Contract C uses `move(uint256, address)`

Wallets would need custom code for every token. Exchanges could not list new tokens without development work. DeFi protocols could not compose with arbitrary tokens.

Standards solve this. When everyone implements the same interface, everything works together.

THE ERC PROCESS

ERC stands for Ethereum Request for Comments. It is the process for proposing standards.

1. Someone writes an EIP (Ethereum Improvement Proposal)
2. The community discusses and refines it
3. If accepted, it becomes an ERC
4. Developers implement the standard
5. The ecosystem builds around it

Major standards:

- ERC-20: Fungible tokens (2015)
- ERC-721: Non-fungible tokens (2018)
- ERC-1155: Multi-token standard (2019)
- ERC-4626: Tokenized vaults (2022)

SECTION 2: ERC-20 FUNGIBLE TOKENS

WHAT IS ERC-20

ERC-20 defines a standard interface for fungible tokens. Fungible means every token is identical -- one USDC is the same as any other USDC.

The standard specifies:

- How to check balances
- How to transfer tokens
- How to approve others to spend your tokens
- Events for tracking transfers and approvals

THE INTERFACE

```
interface IERC20 {  
    function totalSupply() external view returns (uint256);  
    function balanceOf(address account) external view returns  
(uint256);  
    function transfer(address to, uint256 amount) external  
returns (bool);  
    function allowance(address owner, address spender) external  
view returns (uint256);  
    function approve(address spender, uint256 amount) external  
returns (bool);  
    function transferFrom(address from, address to, uint256
```

amount) external returns (bool);

event Transfer(address indexed from, address indexed to,
uint256 value);

event Approval(address indexed owner, address indexed
spender, uint256 value);

}

FUNCTION BY FUNCTION

totalSupply(): Returns the total number of tokens in existence.

balanceOf(address): Returns how many tokens an address holds.

transfer(to, amount): Sends tokens from the caller to the recipient. Returns true on success.

allowance(owner, spender): Returns how many tokens the spender is allowed to spend on behalf of the owner.

approve(spender, amount): Allows the spender to spend up to amount tokens from the caller's balance.

transferFrom(from, to, amount): Transfers tokens from one address to another, using the allowance mechanism. The caller must have been approved.

THE APPROVE/TRANSFERFROM PATTERN

Why have two ways to transfer? Because contracts cannot sign transactions.

When you use a DEX:

1. You approve the DEX to spend your tokens
2. You call the swap function on the DEX

3. The DEX calls transferFrom to take your tokens
4. The DEX sends you the output tokens

Without this pattern, you would need to send tokens to the contract first, creating race conditions and complexity.

COMPLETE ERC-20 IMPLEMENTATION

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns
(uint256);
    function transfer(address to, uint256 amount) external
returns (bool);
    function allowance(address owner, address spender) external
view returns (uint256);
    function approve(address spender, uint256 amount) external
returns (bool);
    function transferFrom(address from, address to, uint256
amount) external returns (bool);

    event Transfer(address indexed from, address indexed to,
uint256 value);
    event Approval(address indexed owner, address indexed
spender, uint256 value);
}

interface IERC20Metadata is IERC20 {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
}
```

```
contract ERC20 is IERC20, IERC20Metadata {  
    string private _name;  
    string private _symbol;  
    uint8 private _decimals;
```

```
    uint256 private _totalSupply;  
    mapping(address => uint256) private _balances;  
    mapping(address => mapping(address => uint256))  
    private _allowances;
```

```
    constructor(string memory name_, string memory symbol_,  
    uint8 decimals_) {  
        _name = name_;  
        _symbol = symbol_;  
        _decimals = decimals_;  
    }
```

```
    function name() public view virtual override returns (string  
memory) {  
        return _name;  
    }
```

```
    function symbol() public view virtual override returns (string  
memory) {  
        return _symbol;  
    }
```

```
    function decimals() public view virtual override returns  
(uint8) {  
        return _decimals;  
    }
```

```
    function totalSupply() public view virtual override returns  
(uint256) {  
        return _totalSupply;  
    }
```

```
function balanceOf(address account) public view virtual
override returns (uint256) {
    return _balances[account];
}
```

```
function transfer(address to, uint256 amount) public virtual
override returns (bool) {
    _transfer(msg.sender, to, amount);
    return true;
}
```

```
function allowance(address owner, address spender) public
view virtual override returns (uint256) {
    return _allowances[owner][spender];
}
```

```
function approve(address spender, uint256 amount) public
virtual override returns (bool) {
    _approve(msg.sender, spender, amount);
    return true;
}
```

```
function transferFrom(address from, address to, uint256
amount) public virtual override returns (bool) {
    _spendAllowance(from, msg.sender, amount);
    _transfer(from, to, amount);
    return true;
}
```

```
function _transfer(address from, address to, uint256 amount)
internal virtual {
    require(from != address(0), "ERC20: transfer from zero
address");
    require(to != address(0), "ERC20: transfer to zero address");

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC20: transfer amount
```

```
exceeds balance");
```

```
unchecked {  
    _balances[from] = fromBalance - amount;  
    _balances[to] += amount;  
}
```

```
emit Transfer(from, to, amount);  
}
```

```
function _approve(address owner, address spender, uint256  
amount) internal virtual {  
    require(owner != address(0), "ERC20: approve from zero  
address");  
    require(spender != address(0), "ERC20: approve to zero  
address");
```

```
    _allowances[owner][spender] = amount;  
    emit Approval(owner, spender, amount);  
}
```

```
function _spendAllowance(address owner, address spender,  
uint256 amount) internal virtual {  
    uint256 currentAllowance = _allowances[owner][spender];  
    if (currentAllowance != type(uint256).max) {  
        require(currentAllowance >= amount, "ERC20: insufficient  
allowance");  
        unchecked {  
            _allowances[owner][spender] = currentAllowance - amount;  
        }  
    }  
}
```

```
function _mint(address account, uint256 amount) internal  
virtual {  
    require(account != address(0), "ERC20: mint to zero  
address");
```

```
_totalSupply += amount;
unchecked {
    _balances[account] += amount;
}
```

```
emit Transfer(address(0), account, amount);
}
```

```
function _burn(address account, uint256 amount) internal
virtual {
    require(account != address(0), "ERC20: burn from zero
address");
```

```
    uint256 accountBalance = _balances[account];
    require(accountBalance >= amount, "ERC20: burn amount
exceeds balance");
```

```
    unchecked {
        _balances[account] = accountBalance - amount;
        _totalSupply -= amount;
    }
```

```
    emit Transfer(account, address(0), amount);
}
}
```

```
contract MyToken is ERC20 {
    address public owner;
```

```
    constructor() ERC20("My Token", "MTK", 18) {
        owner = msg.sender;
        _mint(msg.sender, 1000000 * 10 ** 18);
    }
```

```
    function mint(address to, uint256 amount) public {
        require(msg.sender == owner, "Only owner can mint");
```

```
_mint(to, amount);  
}  
  
function burn(uint256 amount) public {  
    _burn(msg.sender, amount);  
}  
}
```

ERC-20 EXTENSIONS

Common extensions to the base standard:

Mintable: Owner can create new tokens

Burnable: Holders can destroy their tokens

Pausable: Owner can freeze all transfers

Capped: Maximum supply limit

Permit: Gasless approvals via signatures (ERC-2612)

SECTION 3: ERC-721 NON-FUNGIBLE TOKENS

WHAT IS ERC-721

ERC-721 defines non-fungible tokens (NFTs). Each token has a unique ID. Token 1 is different from token 2.

Use cases:

- Digital art
- Collectibles
- Game items
- Domain names
- Real estate deeds
- Tickets and memberships

THE INTERFACE

```
interface IERC721 {
    event Transfer(address indexed from, address indexed to,
uint256 indexed tokenId);
    event Approval(address indexed owner, address indexed
approved, uint256 indexed tokenId);
    event ApprovalForAll(address indexed owner, address
indexed operator, bool approved);

    function balanceOf(address owner) external view returns
(uint256 balance);
    function ownerOf(uint256 tokenId) external view returns
(address owner);

    function safeTransferFrom(address from, address to, uint256
tokenId, bytes calldata data) external;
    function safeTransferFrom(address from, address to, uint256
tokenId) external;
    function transferFrom(address from, address to, uint256
tokenId) external;

    function approve(address to, uint256 tokenId) external;
    function setApprovalForAll(address operator, bool approved)
external;

    function getApproved(uint256 tokenId) external view returns
(address operator);
    function isApprovedForAll(address owner, address operator)
external view returns (bool);
}

interface IERC721Metadata is IERC721 {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function tokenURI(uint256 tokenId) external view returns
(string memory);
```

```
}
```

KEY DIFFERENCES FROM ERC-20

balanceOf: Returns how many NFTs an address owns (count, not amount).

ownerOf: Returns who owns a specific token ID.

tokenURI: Returns metadata location for a specific token.

safeTransferFrom: Checks if receiver can handle NFTs (prevents accidental loss).

setApprovalForAll: Approves an operator for all tokens (useful for marketplaces).

THE SAFE TRANSFER MECHANISM

If you send an ERC-721 to a contract that cannot handle it, the token is lost forever.

safeTransferFrom checks if the recipient is a contract. If so, it calls **onERC721Received** on the recipient. The recipient must return a specific value to confirm it can handle NFTs.

```
interface IERC721Receiver {  
    function onERC721Received(  
        address operator,  
        address from,  
        uint256 tokenId,  
        bytes calldata data  
    ) external returns (bytes4);  
}
```


COMPLETE ERC-721 IMPLEMENTATION

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
interface IERC165 {
    function supportsInterface(bytes4 interfaceId) external view
    returns (bool);
}
```

```
interface IERC721 is IERC165 {
    event Transfer(address indexed from, address indexed to,
    uint256 indexed tokenId);
    event Approval(address indexed owner, address indexed
    approved, uint256 indexed tokenId);
    event ApprovalForAll(address indexed owner, address
    indexed operator, bool approved);
```

```
    function balanceOf(address owner) external view returns
    (uint256);
    function ownerOf(uint256 tokenId) external view returns
    (address);
    function safeTransferFrom(address from, address to, uint256
    tokenId, bytes calldata data) external;
    function safeTransferFrom(address from, address to, uint256
    tokenId) external;
    function transferFrom(address from, address to, uint256
    tokenId) external;
    function approve(address to, uint256 tokenId) external;
    function setApprovalForAll(address operator, bool approved)
    external;
    function getApproved(uint256 tokenId) external view returns
    (address);
    function isApprovedForAll(address owner, address operator)
    external view returns (bool);
}
```

```

interface IERC721Metadata is IERC721 {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function tokenURI(uint256 tokenId) external view returns
(string memory);
}

```

```

interface IERC721Receiver {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}

```

```

contract ERC721 is IERC721, IERC721Metadata {
    string private _name;
    string private _symbol;
    string private _baseURI;

```

```

    mapping(uint256 => address) private _owners;
    mapping(address => uint256) private _balances;
    mapping(uint256 => address) private _tokenApprovals;
    mapping(address => mapping(address => bool)) private
_operatorApprovals;

```

```

    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

```

```

    function supportsInterface(bytes4 interfaceId) public view
virtual override returns (bool) {
    return
        interfaceId == type(IERC721).interfaceId ||
        interfaceId == type(IERC721Metadata).interfaceId ||

```

```
interfaceId == type(IERC165).interfaceId;  
}
```

```
function name() public view virtual override returns (string  
memory) {  
    return _name;  
}
```

```
function symbol() public view virtual override returns (string  
memory) {  
    return _symbol;  
}
```

```
function tokenURI(uint256 tokenId) public view virtual  
override returns (string memory) {  
    require(_exists(tokenId), "ERC721: URI query for nonexistent  
token");  
    return string(abi.encodePacked(_baseURI, _toString(tokenId),  
".json"));  
}
```

```
function balanceOf(address owner) public view virtual  
override returns (uint256) {  
    require(owner != address(0), "ERC721: balance query for  
zero address");  
    return _balances[owner];  
}
```

```
function ownerOf(uint256 tokenId) public view virtual  
override returns (address) {  
    address owner = _owners[tokenId];  
    require(owner != address(0), "ERC721: owner query for  
nonexistent token");  
    return owner;  
}
```

```
function approve(address to, uint256 tokenId) public virtual
```

```

override {
    address owner = ownerOf(tokenId);
    require(to != owner, "ERC721: approval to current owner");
    require(
        msg.sender == owner || isApprovedForAll(owner,
msg.sender),
        "ERC721: approve caller is not owner nor approved for all"
    );
    _tokenApprovals[tokenId] = to;
    emit Approval(owner, to, tokenId);
}

```

```

function getApproved(uint256 tokenId) public view virtual
override returns (address) {
    require(_exists(tokenId), "ERC721: approved query for
nonexistent token");
    return _tokenApprovals[tokenId];
}

```

```

function setApprovalForAll(address operator, bool approved)
public virtual override {
    require(operator != msg.sender, "ERC721: approve to
caller");
    _operatorApprovals[msg.sender][operator] = approved;
    emit ApprovalForAll(msg.sender, operator, approved);
}

```

```

function isApprovedForAll(address owner, address operator)
public view virtual override returns (bool) {
    return _operatorApprovals[owner][operator];
}

```

```

function transferFrom(address from, address to, uint256
tokenId) public virtual override {
    require(_isApprovedOrOwner(msg.sender, tokenId), "ERC721:
transfer caller not owner nor approved");
    _transfer(from, to, tokenId);
}

```

```
}
```

```
function safeTransferFrom(address from, address to, uint256  
tokenId) public virtual override {  
    safeTransferFrom(from, to, tokenId, "");  
}
```

```
function safeTransferFrom(address from, address to, uint256  
tokenId, bytes memory data) public virtual override {  
    require(!_isApprovedOrOwner(msg.sender, tokenId), "ERC721:  
transfer caller not owner nor approved");  
    _safeTransfer(from, to, tokenId, data);  
}
```

```
function _exists(uint256 tokenId) internal view virtual returns  
(bool) {  
    return _owners[tokenId] != address(0);  
}
```

```
function _isApprovedOrOwner(address spender, uint256  
tokenId) internal view virtual returns (bool) {  
    address owner = ownerOf(tokenId);  
    return (spender == owner || getApproved(tokenId) ==  
spender || isApprovedForAll(owner, spender));  
}
```

```
function _mint(address to, uint256 tokenId) internal virtual {  
    require(to != address(0), "ERC721: mint to zero address");  
    require(!_exists(tokenId), "ERC721: token already minted");
```

```
    _balances[to] += 1;  
    _owners[tokenId] = to;
```

```
    emit Transfer(address(0), to, tokenId);  
}
```

```
function _burn(uint256 tokenId) internal virtual {
```

```
address owner = ownerOf(tokenId);
```

```
delete _tokenApprovals[tokenId];
```

```
_balances[owner] -= 1;
```

```
delete _owners[tokenId];
```

```
emit Transfer(owner, address(0), tokenId);
```

```
}
```

```
function _transfer(address from, address to, uint256 tokenId)  
internal virtual {
```

```
    require(ownerOf(tokenId) == from, "ERC721: transfer from  
incorrect owner");
```

```
    require(to != address(0), "ERC721: transfer to zero address");
```

```
delete _tokenApprovals[tokenId];
```

```
_balances[from] -= 1;
```

```
_balances[to] += 1;
```

```
_owners[tokenId] = to;
```

```
emit Transfer(from, to, tokenId);
```

```
}
```

```
function _safeTransfer(address from, address to, uint256  
tokenId, bytes memory data) internal virtual {
```

```
    _transfer(from, to, tokenId);
```

```
    require(
```

```
        _checkOnERC721Received(from, to, tokenId, data),
```

```
        "ERC721: transfer to non ERC721Receiver implementer"
```

```
    );
```

```
}
```

```
function _checkOnERC721Received(
```

```
    address from,
```

```
    address to,
```

```
    uint256 tokenId,
```

```

bytes memory data
) private returns (bool) {
if (to.code.length > 0) {
try IERC721Receiver(to).onERC721Received(msg.sender,
from, tokenId, data) returns (bytes4 retval) {
return retval ==
IERC721Receiver.onERC721Received.selector;
} catch (bytes memory reason) {
if (reason.length == 0) {
revert("ERC721: transfer to non ERC721Receiver
implementer");
} else {
assembly {
revert(add(32, reason), mload(reason))
}
}
} else {
return true;
}
}
}

```

```

function _setBaseURI(string memory baseURI_) internal
virtual {
_baseURI = baseURI_;
}

```

```

function _toString(uint256 value) internal pure returns (string
memory) {
if (value == 0) {
return "0";
}
uint256 temp = value;
uint256 digits;
while (temp != 0) {
digits++;
temp /= 10;
}
}

```

```

}
bytes memory buffer = new bytes(digits);
while (value != 0) {
    digits -= 1;
    buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
    value /= 10;
}
return string(buffer);
}
}

```

```

contract MyNFT is ERC721 {
    address public owner;
    uint256 public totalSupply;
    uint256 public maxSupply;
    uint256 public mintPrice;

```

```

    constructor() ERC721("My NFT Collection", "MNFT") {
        owner = msg.sender;
        maxSupply = 10000;
        mintPrice = 0.05 ether;
        _setBaseURI("https://api.mynft.com/metadata/");
    }

```

```

    function mint(uint256 quantity) public payable {
        require(totalSupply + quantity <= maxSupply, "Exceeds
max supply");
        require(msg.value >= mintPrice * quantity, "Insufficient
payment");

```

```

    for (uint256 i = 0; i < quantity; i++) {
        _mint(msg.sender, totalSupply);
        totalSupply++;
    }
}

```

```

    function withdraw() public {

```



```
require(msg.sender == owner, "Only owner");  
payable(owner).transfer(address(this).balance);  
}  
}
```

SECTION 4: ERC-1155 MULTI-TOKEN STANDARD

WHAT IS ERC-1155

ERC-1155 combines fungible and non-fungible tokens in a single contract. One contract can manage:

- Fungible tokens (like ERC-20)
- Non-fungible tokens (like ERC-721)
- Semi-fungible tokens (fungible within a type)

Designed for games: swords, potions, unique items, currencies
-- all in one contract.

EFFICIENCY BENEFITS

Batch transfers: Move multiple token types in one transaction.

Shared logic: One contract, one deployment, lower gas.

Atomic swaps: Exchange multiple items atomically.

THE INTERFACE

```
interface IERC1155 {  
    event TransferSingle(  
        address indexed operator,  
        address indexed from,  
        address indexed to,  
        uint256 id,  
        uint256 value
```

```
);
```

```
event TransferBatch(  
    address indexed operator,  
    address indexed from,  
    address indexed to,  
    uint256[] ids,  
    uint256[] values  
);
```

```
event ApprovalForAll(address indexed account, address  
indexed operator, bool approved);  
event URI(string value, uint256 indexed id);
```

```
function balanceOf(address account, uint256 id) external  
view returns (uint256);
```

```
function balanceOfBatch(address[] calldata accounts,  
uint256[] calldata ids) external view returns (uint256[]  
memory);
```

```
function setApprovalForAll(address operator, bool approved)  
external;
```

```
function isApprovedForAll(address account, address operator)  
external view returns (bool);
```

```
function safeTransferFrom(address from, address to, uint256  
id, uint256 amount, bytes calldata data) external;
```

```
function safeBatchTransferFrom(address from, address to,  
uint256[] calldata ids, uint256[] calldata amounts, bytes  
calldata data) external;
```

```
}
```

COMPLETE ERC-1155 IMPLEMENTATION

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
interface IERC1155Receiver {
```

```
function onERC1155Received(
address operator,
address from,
uint256 id,
uint256 value,
bytes calldata data
) external returns (bytes4);
```

```
function onERC1155BatchReceived(
address operator,
address from,
uint256[] calldata ids,
uint256[] calldata values,
bytes calldata data
) external returns (bytes4);
}
```

```
contract ERC1155 {
mapping(uint256 => mapping(address => uint256))
private _balances;
mapping(address => mapping(address => bool)) private
_operatorApprovals;
string private _uri;
```

```
event TransferSingle(address indexed operator, address
indexed from, address indexed to, uint256 id, uint256 value);
event TransferBatch(address indexed operator, address
indexed from, address indexed to, uint256[] ids, uint256[]
values);
event ApprovalForAll(address indexed account, address
indexed operator, bool approved);
event URI(string value, uint256 indexed id);
```

```
constructor(string memory uri_) {
_uri = uri_;
}
```

```
function uri(uint256) public view virtual returns (string
memory) {
    return _uri;
}
```

```
function balanceOf(address account, uint256 id) public view
virtual returns (uint256) {
    require(account != address(0), "ERC1155: balance query for
zero address");
    return _balances[id][account];
}
```

```
function balanceOfBatch(
    address[] memory accounts,
    uint256[] memory ids
) public view virtual returns (uint256[] memory) {
    require(accounts.length == ids.length, "ERC1155: accounts
and ids length mismatch");
```

```
    uint256[] memory batchBalances = new uint256[]
(accounts.length);
    for (uint256 i = 0; i < accounts.length; ++i) {
        batchBalances[i] = balanceOf(accounts[i], ids[i]);
    }
    return batchBalances;
}
```

```
function setApprovalForAll(address operator, bool approved)
public virtual {
    require(msg.sender != operator, "ERC1155: setting approval
status for self");
    _operatorApprovals[msg.sender][operator] = approved;
    emit ApprovalForAll(msg.sender, operator, approved);
}
```

```
function isApprovedForAll(address account, address operator)
public view virtual returns (bool) {
```

```
return _operatorApprovals[account][operator];  
}
```

```
function safeTransferFrom(  
    address from,  
    address to,  
    uint256 id,  
    uint256 amount,  
    bytes memory data  
) public virtual {  
    require(  
        from == msg.sender || isApprovedForAll(from, msg.sender),  
        "ERC1155: caller is not owner nor approved"  
    );  
    _safeTransferFrom(from, to, id, amount, data);  
}
```

```
function safeBatchTransferFrom(  
    address from,  
    address to,  
    uint256[] memory ids,  
    uint256[] memory amounts,  
    bytes memory data  
) public virtual {  
    require(  
        from == msg.sender || isApprovedForAll(from, msg.sender),  
        "ERC1155: caller is not owner nor approved"  
    );  
    _safeBatchTransferFrom(from, to, ids, amounts, data);  
}
```

```
function _safeTransferFrom(  
    address from,  
    address to,  
    uint256 id,  
    uint256 amount,  
    bytes memory data
```

```

) internal virtual {
    require(to != address(0), "ERC1155: transfer to zero
address");

    uint256 fromBalance = _balances[id][from];
    require(fromBalance >= amount, "ERC1155: insufficient
balance for transfer");

    unchecked {
        _balances[id][from] = fromBalance - amount;
    }
    _balances[id][to] += amount;

    emit TransferSingle(msg.sender, from, to, id, amount);

    _doSafeTransferAcceptanceCheck(msg.sender, from, to, id,
amount, data);
}

function _safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    require(ids.length == amounts.length, "ERC1155: ids and
amounts length mismatch");
    require(to != address(0), "ERC1155: transfer to zero
address");

    for (uint256 i = 0; i < ids.length; ++i) {
        uint256 id = ids[i];
        uint256 amount = amounts[i];

        uint256 fromBalance = _balances[id][from];
        require(fromBalance >= amount, "ERC1155: insufficient

```

```
balance for transfer");
```

```
unchecked {  
  _balances[id][from] = fromBalance - amount;  
}  
_balances[id][to] += amount;  
}
```

```
emit TransferBatch(msg.sender, from, to, ids, amounts);
```

```
_doSafeBatchTransferAcceptanceCheck(msg.sender, from, to,  
ids, amounts, data);  
}
```

```
function _mint(address to, uint256 id, uint256 amount, bytes  
memory data) internal virtual {  
  require(to != address(0), "ERC1155: mint to zero address");
```

```
_balances[id][to] += amount;  
emit TransferSingle(msg.sender, address(0), to, id, amount);
```

```
_doSafeTransferAcceptanceCheck(msg.sender, address(0), to,  
id, amount, data);  
}
```

```
function _mintBatch(  
  address to,  
  uint256[] memory ids,  
  uint256[] memory amounts,  
  bytes memory data  
) internal virtual {  
  require(to != address(0), "ERC1155: mint to zero address");  
  require(ids.length == amounts.length, "ERC1155: ids and  
amounts length mismatch");
```

```
  for (uint256 i = 0; i < ids.length; i++) {  
    _balances[ids[i]][to] += amounts[i];
```

```
}
```

```
emit TransferBatch(msg.sender, address(0), to, ids, amounts);
```

```
_doSafeBatchTransferAcceptanceCheck(msg.sender,  
address(0), to, ids, amounts, data);
```

```
}
```

```
function _burn(address from, uint256 id, uint256 amount)
```

```
internal virtual {
```

```
    require(from != address(0), "ERC1155: burn from zero  
address");
```

```
    uint256 fromBalance = _balances[id][from];
```

```
    require(fromBalance >= amount, "ERC1155: burn amount  
exceeds balance");
```

```
    unchecked {
```

```
        _balances[id][from] = fromBalance - amount;
```

```
    }
```

```
    emit TransferSingle(msg.sender, from, address(0), id,  
amount);
```

```
}
```

```
function _doSafeTransferAcceptanceCheck(
```

```
    address operator,
```

```
    address from,
```

```
    address to,
```

```
    uint256 id,
```

```
    uint256 amount,
```

```
    bytes memory data
```

```
) private {
```

```
    if (to.code.length > 0) {
```

```
        try IERC1155Receiver(to).onERC1155Received(operator,  
from, id, amount, data) returns (bytes4 response) {
```

```
        if (response !=
```



```

IERC1155Receiver.onERC1155Received.selector) {
    revert("ERC1155: ERC1155Receiver rejected tokens");
}
} catch Error(string memory reason) {
    revert(reason);
} catch {
    revert("ERC1155: transfer to non ERC1155Receiver
implementer");
}
}
}

function _doSafeBatchTransferAcceptanceCheck(
    address operator,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) private {
    if (to.code.length > 0) {
        try
            IERC1155Receiver(to).onERC1155BatchReceived(operator,
            from, ids, amounts, data) returns (bytes4 response) {
                if (response !=
                IERC1155Receiver.onERC1155BatchReceived.selector) {
                    revert("ERC1155: ERC1155Receiver rejected tokens");
                }
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("ERC1155: transfer to non ERC1155Receiver
                implementer");
            }
        }
    }
}

```

```

contract GameItems is ERC1155 {
    uint256 public constant GOLD = 0;
    uint256 public constant SILVER = 1;
    uint256 public constant SWORD = 2;
    uint256 public constant SHIELD = 3;
    uint256 public constant LEGENDARY_SWORD = 4;

    address public owner;

    constructor() ERC1155("https://game.example/api/item/
{id}.json") {
        owner = msg.sender;

        _mint(msg.sender, GOLD, 10**18, "");
        _mint(msg.sender, SILVER, 10**18, "");
        _mint(msg.sender, SWORD, 1000, "");
        _mint(msg.sender, SHIELD, 500, "");
        _mint(msg.sender, LEGENDARY_SWORD, 1, "");
    }

    function mint(address to, uint256 id, uint256 amount) public
    {
        require(msg.sender == owner, "Only owner");
        _mint(to, id, amount, "");
    }

    function mintBatch(address to, uint256[] memory ids,
    uint256[] memory amounts) public {
        require(msg.sender == owner, "Only owner");
        _mintBatch(to, ids, amounts, "");
    }

    function burn(uint256 id, uint256 amount) public {
        _burn(msg.sender, id, amount);
    }
}

```

SECTION 5: PROXY PATTERNS

WHY PROXIES

Smart contracts are immutable. Once deployed, the code cannot change. But requirements change. Bugs are discovered. Features are added.

Proxies solve this by separating:

- The proxy (stores data, never changes)
- The implementation (contains logic, can be replaced)

HOW PROXIES WORK

1. User calls the proxy
2. Proxy delegates the call to the implementation
3. Implementation executes using the proxy's storage
4. Result returns to user

The key is `delegatecall`: execute another contract's code in the caller's context.

THE TRANSPARENT PROXY PATTERN

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
contract TransparentProxy {
    bytes32 private constant IMPLEMENTATION_SLOT =
        bytes32(uint256(keccak256("eip1967.proxy.implementation")))
        - 1);
    bytes32 private constant ADMIN_SLOT =
```

```
bytes32(uint256(keccak256("eip1967.proxy.admin")) - 1);
```

```
constructor(address implementation, address admin) {  
  _setImplementation(implementation);  
  _setAdmin(admin);  
}
```

```
modifier onlyAdmin() {  
  require(msg.sender == _getAdmin(), "Not admin");  
  _;  
}
```

```
function upgradeTo(address newImplementation) external  
onlyAdmin {  
  _setImplementation(newImplementation);  
}
```

```
function implementation() external view returns (address) {  
  return _getImplementation();  
}
```

```
function admin() external view returns (address) {  
  return _getAdmin();  
}
```

```
function _getImplementation() internal view returns (address  
impl) {  
  bytes32 slot = IMPLEMENTATION_SLOT;  
  assembly {  
    impl := sload(slot)  
  }  
}
```

```
function _setImplementation(address newImplementation)  
internal {  
  bytes32 slot = IMPLEMENTATION_SLOT;  
  assembly {
```

```
sstore(slot, newImplementation)
}
}
```

```
function _getAdmin() internal view returns (address adm) {
bytes32 slot = ADMIN_SLOT;
assembly {
adm := sload(slot)
}
}
```

```
function _setAdmin(address newAdmin) internal {
bytes32 slot = ADMIN_SLOT;
assembly {
sstore(slot, newAdmin)
}
}
```

```
fallback() external payable {
address impl = _getImplementation();
assembly {
calldatacopy(0, 0, calldatasize())
let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
returndatacopy(0, 0, returndatasize())
}
```

```
switch result
case 0 {
revert(0, returndatasize())
}
default {
return(0, returndatasize())
}
}
}
```

```
receive() external payable {}
}
```

THE UUPS PATTERN

UUPS (Universal Upgradeable Proxy Standard) puts upgrade logic in the implementation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
abstract contract UUPSUpgradeable {
    bytes32 private constant IMPLEMENTATION_SLOT =
    bytes32(uint256(keccak256("eip1967.proxy.implementation"))
    - 1);
```

```
    function upgradeTo(address newImplementation) public
    virtual {
        _authorizeUpgrade(newImplementation);
        _setImplementation(newImplementation);
    }
```

```
    function _authorizeUpgrade(address newImplementation)
    internal virtual;
```

```
    function _setImplementation(address newImplementation)
    internal {
        bytes32 slot = IMPLEMENTATION_SLOT;
        assembly {
            sstore(slot, newImplementation)
        }
    }
```

```
    function _getImplementation() internal view returns (address
    impl) {
        bytes32 slot = IMPLEMENTATION_SLOT;
        assembly {
            impl := sload(slot)
        }
    }
```

```
}  
}  
}
```

```
contract MyContractV1 is UUPSUpgradeable {  
    address public owner;  
    uint256 public value;  
  
    function initialize(address _owner) public {  
        require(owner == address(0), "Already initialized");  
        owner = _owner;  
    }  
  
    function setValue(uint256 _value) public {  
        value = _value;  
    }  
  
    function _authorizeUpgrade(address) internal override {  
        require(msg.sender == owner, "Not owner");  
    }  
}
```

```
contract MyContractV2 is UUPSUpgradeable {  
    address public owner;  
    uint256 public value;  
    uint256 public newFeature;  
  
    function setValue(uint256 _value) public {  
        value = _value * 2;  
    }  
  
    function setNewFeature(uint256 _value) public {  
        newFeature = _value;  
    }  
  
    function _authorizeUpgrade(address) internal override {  
        require(msg.sender == owner, "Not owner");  
    }  
}
```

```
}  
}
```

PROXY SAFETY RULES

1. Never use constructors in implementation contracts (use initialize functions)
2. Never selfdestruct in implementation contracts
3. Maintain storage layout compatibility between versions
4. Always test upgrades thoroughly

SECTION 6: FACTORY PATTERNS

WHAT IS A FACTORY

A factory contract creates other contracts. Use cases:

- Deploying token contracts for each user
- Creating pairs in a DEX
- Spawning game instances

BASIC FACTORY

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.20;
```

```
contract Token {  
    string public name;  
    string public symbol;  
    address public owner;  
    uint256 public totalSupply;  
    mapping(address => uint256) public balanceOf;  
  
    constructor(string memory _name, string memory _symbol,
```



```
uint256 _initialSupply, address _owner) {  
    name = _name;  
    symbol = _symbol;  
    owner = _owner;  
    totalSupply = _initialSupply;  
    balanceOf[_owner] = _initialSupply;  
}
```

```
function transfer(address to, uint256 amount) public returns  
(bool) {  
    require(balanceOf[msg.sender] >= amount, "Insufficient  
balance");  
    balanceOf[msg.sender] -= amount;  
    balanceOf[to] += amount;  
    return true;  
}  
}
```

```
contract TokenFactory {  
    address[] public tokens;  
    mapping(address => address[]) public tokensByCreator;
```

```
    event TokenCreated(address indexed token, address indexed  
creator, string name, string symbol);
```

```
    function createToken(  
        string memory name,  
        string memory symbol,  
        uint256 initialSupply  
    ) public returns (address) {  
        Token token = new Token(name, symbol, initialSupply,  
msg.sender);  
        address tokenAddress = address(token);  
  
        tokens.push(tokenAddress);  
        tokensByCreator[msg.sender].push(tokenAddress);
```

```

    emit TokenCreated(tokenAddress, msg.sender, name,
symbol);

    return tokenAddress;
}

function getTokenCount() public view returns (uint256) {
    return tokens.length;
}

function getTokensByCreator(address creator) public view
returns (address[] memory) {
    return tokensByCreator[creator];
}
}

```

CREATE2 FOR DETERMINISTIC ADDRESSES

create2 deploys to a predictable address based on:

- Deployer address
- Salt (arbitrary bytes32)
- Bytecode hash

```

contract Create2Factory {
    event Deployed(address addr, bytes32 salt);

    function deploy(bytes32 salt, bytes memory bytecode) public
returns (address) {
    address addr;
    assembly {
        addr := create2(0, add(bytecode, 0x20), mload(bytecode),
salt)
        if iszero(extcodesize(addr)) {
            revert(0, 0)
        }
    }
}
}

```

```

emit Deployed(addr, salt);
return addr;
}

```

```

function computeAddress(bytes32 salt, bytes32
bytecodeHash) public view returns (address) {
    return
    address(uint160(uint256(keccak256(abi.encodePacked(
        bytes1(0xff),
        address(this),
        salt,
        bytecodeHash
    )))));
}
}

```

CLONE FACTORY (MINIMAL PROXY)

Clones are cheap copies that delegate to an implementation:

```

contract CloneFactory {
    function createClone(address implementation) internal
returns (address instance) {
    assembly {
        let ptr := mload(0x40)
        mstore(ptr,
0x3d602d80600a3d3981f3363d3d373d3d3d363d7300000000000000
        mstore(add(ptr, 0x14), shl(0x60, implementation))
        mstore(add(ptr, 0x28),
0x5af43d82803e903d91602b57fd5bf3000000000000000000000000
        instance := create(0, ptr, 0x37)
    }
    require(instance != address(0), "Clone creation failed");
}
}

```

SECTION 7: ACCESS CONTROL

BASIC OWNERSHIP

```
contract Ownable {
    address public owner;

    event OwnershipTransferred(address indexed previousOwner,
    address indexed newOwner);

    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    function transferOwnership(address newOwner) public
    onlyOwner {
        require(newOwner != address(0), "Zero address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(owner, address(0));
        owner = address(0);
    }
}
```

ROLE-BASED ACCESS CONTROL

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
contract AccessControl {
    struct RoleData {
        mapping(address => bool) members;
        bytes32 adminRole;
    }
```

```
    mapping(bytes32 => RoleData) private _roles;
```

```
    bytes32 public constant DEFAULT_ADMIN_ROLE = 0x00;
    bytes32 public constant MINTER_ROLE =
    keccak256("MINTER_ROLE");
    bytes32 public constant PAUSER_ROLE =
    keccak256("PAUSER_ROLE");
    bytes32 public constant BURNER_ROLE =
    keccak256("BURNER_ROLE");
```

```
    event RoleGranted(bytes32 indexed role, address indexed
    account, address indexed sender);
    event RoleRevoked(bytes32 indexed role, address indexed
    account, address indexed sender);
    event RoleAdminChanged(bytes32 indexed role, bytes32
    indexed previousAdminRole, bytes32 indexed
    newAdminRole);
```

```
    constructor() {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }
```

```
    modifier onlyRole(bytes32 role) {
        require(hasRole(role, msg.sender), "AccessControl: missing
        role");
    }
```

```
function hasRole(bytes32 role, address account) public view  
returns (bool) {  
    return _roles[role].members[account];  
}
```

```
function getRoleAdmin(bytes32 role) public view returns  
(bytes32) {  
    return _roles[role].adminRole;  
}
```

```
function grantRole(bytes32 role, address account) public  
onlyRole(getRoleAdmin(role)) {  
    _grantRole(role, account);  
}
```

```
function revokeRole(bytes32 role, address account) public  
onlyRole(getRoleAdmin(role)) {  
    _revokeRole(role, account);  
}
```

```
function renounceRole(bytes32 role, address account) public  
{  
    require(account == msg.sender, "AccessControl: can only  
renounce roles for self");  
    _revokeRole(role, account);  
}
```

```
function _grantRole(bytes32 role, address account) internal {  
    if (!hasRole(role, account)) {  
        _roles[role].members[account] = true;  
        emit RoleGranted(role, account, msg.sender);  
    }  
}
```

```
function _revokeRole(bytes32 role, address account) internal  
{
```

```

if (hasRole(role, account)) {
    _roles[role].members[account] = false;
    emit RoleRevoked(role, account, msg.sender);
}
}

```

```

function _setRoleAdmin(bytes32 role, bytes32 adminRole)
internal {
    bytes32 previousAdminRole = getRoleAdmin(role);
    _roles[role].adminRole = adminRole;
    emit RoleAdminChanged(role, previousAdminRole,
adminRole);
}
}

```

```

contract ManagedToken is AccessControl {
    string public name = "Managed Token";
    string public symbol = "MGT";
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    bool public paused;

```

```

constructor() {
    _grantRole(MINTER_ROLE, msg.sender);
    _grantRole(PAUSER_ROLE, msg.sender);
    _grantRole(BURNER_ROLE, msg.sender);
}

```

```

modifier whenNotPaused() {
    require(!paused, "Paused");
    _;
}

```

```

function mint(address to, uint256 amount) public
onlyRole(MINTER_ROLE) whenNotPaused {
    totalSupply += amount;
    balanceOf[to] += amount;
}

```

```

}

function burn(address from, uint256 amount) public
onlyRole(BURNER_ROLE) {
    require(balanceOf[from] >= amount, "Insufficient balance");
    totalSupply -= amount;
    balanceOf[from] -= amount;
}

function pause() public onlyRole(PAUSER_ROLE) {
    paused = true;
}

function unpause() public onlyRole(PAUSER_ROLE) {
    paused = false;
}

function transfer(address to, uint256 amount) public
whenNotPaused returns (bool) {
    require(balanceOf[msg.sender] >= amount, "Insufficient
balance");
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
    return true;
}
}

```

SECTION 8: WHAT YOU LEARNED

After reading this chapter, you understand:

1. Why standards matter: Interoperability, composability, ecosystem compatibility
2. ERC-20: Fungible tokens, approve/transferFrom pattern,

extensions

3. ERC-721: Non-fungible tokens, safe transfers, metadata
4. ERC-1155: Multi-token standard, batch operations, game items
5. Proxy patterns: Transparent proxy, UUPS, upgrade safety
6. Factory patterns: Basic factory, CREATE2, minimal clones
7. Access control: Ownership, role-based access, granular permissions

You have complete implementations of:

- ERC-20 token with mint and burn
- ERC-721 NFT collection with minting
- ERC-1155 game items
- Transparent and UUPS proxies
- Token factory
- Role-based access control

These standards and patterns are the building blocks of all blockchain applications. DeFi protocols combine ERC-20 with custom logic. NFT marketplaces use ERC-721 and ERC-1155. Every upgradeable contract uses a proxy pattern.

In the next chapter, we explore wallet integration: connecting frontend applications to blockchain wallets using MetaMask, WalletConnect, and other protocols.

CHAPTER 8: WALLET INTEGRATION

The bridge between users and blockchain applications is the wallet. Without wallet integration your smart contracts sit unused on the blockchain waiting for someone to interact with them. This chapter covers everything you need to know about connecting wallets to your applications from

understanding wallet types to implementing production-ready integration patterns.

SECTION 1: UNDERSTANDING WALLET TYPES

What is a Cryptocurrency Wallet

A wallet is software or hardware that manages private keys and enables users to sign transactions. The wallet does not actually store cryptocurrency. Your tokens and coins live on the blockchain. The wallet stores the private keys that prove you own those assets and can authorize transfers.

Think of it like your house key. The key does not contain your house but it proves you have the right to enter. Your wallet key does not contain your crypto but it proves you have the right to spend it.

Every wallet address is derived from a private key through a one-way mathematical function. You can generate the address from the private key but you cannot derive the private key from the address. This asymmetry is fundamental to blockchain security.

Custodial vs Non-Custodial

The first major distinction in wallet types is who controls the private keys.

Custodial wallets are managed by a third party. When you create an account on Coinbase or Binance your keys are stored on their servers. You access your funds through username and password authentication. The exchange holds your actual private keys.

Advantages of custodial wallets include password recovery if

you forget credentials account protection through the platform and simplified user experience. Disadvantages include counterparty risk if the exchange is hacked or goes bankrupt you lose access. You also cannot interact directly with DeFi protocols in most cases.

Non-custodial wallets put you in complete control of your private keys. MetaMask Trust Wallet and hardware wallets all fall into this category. You and only you have access to your keys.

Advantages of non-custodial include true ownership nobody can freeze your funds direct access to all blockchain applications and privacy. Disadvantages include full responsibility if you lose your seed phrase your funds are gone forever with no recovery option.

For blockchain applications you will almost exclusively integrate with non-custodial wallets because users need to sign transactions directly.

Hot Wallets vs Cold Wallets

The second distinction is whether the wallet connects to the internet.

Hot wallets are connected to the internet. Browser extensions like MetaMask mobile apps like Trust Wallet and desktop applications like Exodus are all hot wallets. They are convenient for daily transactions but more vulnerable to hacking.

Cold wallets remain offline. Hardware wallets like Ledger and Trezor store keys on a physical device that only connects when you need to sign a transaction. Paper wallets are literally printed private keys stored offline.

Cold storage is recommended for large holdings while hot wallets are suitable for funds you actively use. Many users maintain both using hot wallets for DeFi activities and cold storage for long-term holdings.

Browser Extension Wallets

Browser extension wallets revolutionized Web3 by injecting a wallet interface directly into web pages. MetaMask pioneered this approach in 2016 and remains the dominant browser wallet.

When installed a browser extension wallet injects a JavaScript object into every page you visit. On Ethereum this object is `window.ethereum`. Applications detect this object to know a wallet is available.

The injected provider follows a standard interface defined by EIP-1193. This standardization means applications can work with any compatible wallet not just MetaMask. Rabby Wallet Coinbase Wallet and Frame all inject compatible providers.

Browser wallets store encrypted keys in browser storage. When you enter your password the wallet decrypts your keys into memory allowing transaction signing. Keys are never exposed to websites. Sites can only request signatures which the wallet displays for user approval.

Mobile Wallets

Mobile wallets run as applications on iOS and Android. Trust Wallet Rainbow Wallet and MetaMask Mobile are popular choices. They provide the same functionality as browser extensions but on mobile devices.

Mobile wallets face a unique challenge. Mobile browsers do not support extensions. Connecting a mobile wallet to a decentralized application requires different approaches.

Deep linking allows applications to open wallet apps directly passing connection and transaction data. When you click a connect button the app can open MetaMask Mobile with the connection request pre-filled.

WalletConnect protocol enables QR code scanning. A desktop application displays a QR code that your mobile wallet scans. This establishes an encrypted connection allowing the desktop app to request signatures from your phone.

Mobile wallets also include built-in browsers. MetaMask Mobile has a DApp browser that works like a desktop browser with the wallet already injected. This provides the smoothest mobile experience for Web3 applications.

Hardware Wallets

Hardware wallets are dedicated physical devices that store private keys securely. Ledger uses a secure element chip similar to credit cards. Trezor uses a general microcontroller with security-focused firmware.

The key security feature is that private keys never leave the device. When you sign a transaction the transaction data is sent to the hardware wallet which signs it internally and returns only the signature. Even if your computer is compromised an attacker cannot extract your keys.

Hardware wallets connect to software interfaces. Ledger Live is Ledger's official application. Hardware wallets also work with MetaMask through USB or Bluetooth connection. When you connect a hardware wallet to MetaMask you get the

convenience of browser integration with the security of hardware key storage.

The tradeoff is user experience. Each transaction requires physical interaction with the device adding friction to frequent trading. Hardware wallets are best for securing significant holdings rather than active DeFi participation.

Smart Contract Wallets

Traditional wallets are Externally Owned Accounts or EOAs. They are controlled by a private key and can only execute transactions the owner signs.

Smart contract wallets are accounts controlled by smart contract code rather than a private key. This enables features impossible with EOAs.

Multi-signature requirements can demand multiple parties approve transactions. Social recovery allows designated guardians to help recover access if you lose your key. Spending limits can restrict daily transaction amounts. Session keys can authorize limited actions without full wallet access.

Safe (formerly Gnosis Safe) is the most widely used smart contract wallet securing over \$100 billion in assets. Account abstraction through ERC-4337 is bringing smart wallet features to the protocol level with wallets like Kernel and ZeroDev.

Smart contract wallets interact differently with applications. They cannot sign messages the same way EOAs do because they have no private key. Standards like EIP-1271 define how smart wallets validate signatures.

SECTION 2: METAMASK INTEGRATION

The Injected Provider

MetaMask injects the ethereum object into the browser's window namespace. This provider follows EIP-1193 and is your interface to the user's wallet.

The following JavaScript checks if MetaMask or a compatible wallet is available.

```
async function checkWalletAvailable() {  
  if (typeof window.ethereum === 'undefined') {  
    return { available: false, message: 'No wallet detected' };  
  }  
  
  const isMetaMask = window.ethereum.isMetaMask;  
  
  return {  
    available: true,  
    isMetaMask: isMetaMask,  
    message: isMetaMask ? 'MetaMask detected' : 'Compatible  
wallet detected'  
  };  
}
```

The window.ethereum object exists if any EIP-1193 compatible wallet is installed. The isMetaMask property specifically identifies MetaMask. Multiple wallets can be installed simultaneously which creates challenges we will address later.

Requesting Account Access

Before your application can see user addresses or request signatures you must request permission. This uses the

eth_requestAccounts RPC method.

The following function requests wallet connection and returns connected accounts.

```
async function connectWallet() {  
  if (typeof window.ethereum === 'undefined') {  
    throw new Error('No Ethereum wallet detected');  
  }  
  
  const accounts = await window.ethereum.request({  
    method: 'eth_requestAccounts'  
  });  
  
  if (accounts.length === 0) {  
    throw new Error('No accounts returned');  
  }  
  
  return accounts[0];  
}
```

When you call eth_requestAccounts MetaMask displays a popup asking the user to connect. They can select which accounts to share with your site. Once approved the accounts array contains their selected addresses.

The first account in the array is typically the user's active account. This is the account that will sign transactions by default.

Detecting Account and Network Changes

Users can switch accounts or networks while using your application. You must listen for these changes to keep your application state synchronized.

The following code sets up event listeners for wallet changes.

```
function setupWalletListeners(onAccountChange,  
onChainChange) {  
  if (typeof window.ethereum === 'undefined') {  
    return;  
  }  
  
  window.ethereum.on('accountsChanged', (accounts) => {  
    if (accounts.length === 0) {  
      onAccountChange(null);  
    } else {  
      onAccountChange(accounts[0]);  
    }  
  });  
  
  window.ethereum.on('chainChanged', (chainId) => {  
    onChainChange(chainId);  
  });  
  
  window.ethereum.on('disconnect', (error) => {  
    console.log('Wallet disconnected', error);  
    onAccountChange(null);  
  });  
}
```

The accountsChanged event fires when the user switches accounts in MetaMask. If the array is empty they have disconnected your site. The chainChanged event fires when they switch networks. Many applications reload the page on chain change to reset state cleanly.

Reading Account Balance

Once connected you can read the account balance using `eth_getBalance`. The result is returned in wei as a hexadecimal

string.

The following function retrieves and formats the balance.

```
async function getBalance(address) {  
  const balanceHex = await window.ethereum.request({  
    method: 'eth_getBalance',  
    params: [address, 'latest']  
  });  
  
  const balanceWei = BigInt(balanceHex);  
  const balanceEth = Number(balanceWei) / 1e18;  
  
  return {  
    wei: balanceWei.toString(),  
    ether: balanceEth.toFixed(6)  
  };  
}
```

The second parameter 'latest' specifies we want the balance at the latest block. You can also use 'pending' for pending state or a specific block number.

Sending Transactions

To send a transaction you construct a transaction object and submit it through `eth_sendTransaction`. MetaMask displays the transaction for user approval.

The following function sends ETH to an address.

```
async function sendTransaction(toAddress, amountEther) {  
  const accounts = await window.ethereum.request({  
    method: 'eth_accounts'  
  });  
}
```

```

if (accounts.length === 0) {
  throw new Error('No connected account');
}

const amountWei = BigInt(Math.floor(amountEther * 1e18));
const amountHex = '0x' + amountWei.toString(16);

const transactionParameters = {
  from: accounts[0],
  to: toAddress,
  value: amountHex
};

const txHash = await window.ethereum.request({
  method: 'eth_sendTransaction',
  params: [transactionParameters]
});

return txHash;
}

```

MetaMask automatically estimates gas for simple transfers. For contract interactions you may need to specify gas limits. The function returns a transaction hash immediately. The transaction is not yet confirmed just submitted to the network.

Signing Messages

Message signing allows users to prove they control an address without sending a transaction. This is commonly used for authentication.

The following function requests a personal signature.

```

async function signMessage(message) {
  const accounts = await window.ethereum.request({

```

```

method: 'eth_accounts'
});

if (accounts.length === 0) {
  throw new Error('No connected account');
}

const signature = await window.ethereum.request({
  method: 'personal_sign',
  params: [message, accounts[0]]
});

return signature;
}

```

The `personal_sign` method prepends a standard prefix to the message before signing. This prefix includes the message length and prevents signed messages from being valid transactions. MetaMask displays the message to the user so they know what they are signing.

Typed Data Signing (EIP-712)

EIP-712 defines a standard for signing structured data. Instead of signing arbitrary strings users sign typed objects with clear field names and values.

The following function signs typed data following EIP-712.

```

async function signTypedData(domain, types, value) {
  const accounts = await window.ethereum.request({
    method: 'eth_accounts'
  });

  if (accounts.length === 0) {
    throw new Error('No connected account');
  }
}

```

```

}

const typedData = {
  types: {
    EIP712Domain: [
      { name: 'name', type: 'string' },
      { name: 'version', type: 'string' },
      { name: 'chainId', type: 'uint256' },
      { name: 'verifyingContract', type: 'address' }
    ],
    ...types
  },
  primaryType: Object.keys(types)[0],
  domain: domain,
  message: value
};

const signature = await window.ethereum.request({
  method: 'eth_signTypedData_v4',
  params: [accounts[0], JSON.stringify(typedData)]
});

return signature;
}

```

The domain object identifies your application and prevents signatures from being replayed across different applications. The types object defines the structure of what is being signed. MetaMask displays this structured data clearly to users making it obvious what they are approving.

Adding Custom Networks

If your application uses a network not in MetaMask by default you can request to add it. This uses `wallet_addEthereumChain`.

The following function adds Polygon network to MetaMask.

```
async function addPolygonNetwork() {  
  const polygonParams = {  
    chainId: '0x89',  
    chainName: 'Polygon Mainnet',  
    nativeCurrency: {  
      name: 'MATIC',  
      symbol: 'MATIC',  
      decimals: 18  
    },  
    rpcUrls: ['https://polygon-rpc.com'],  
    blockExplorerUrls: ['https://polygonscan.com']  
  };  
  
  await window.ethereum.request({  
    method: 'wallet_addEthereumChain',  
    params: [polygonParams]  
  });  
}
```

If the network already exists in MetaMask's custom networks this prompts the user to switch to it. The chainId must be hexadecimal with 0x prefix.

Switching Networks

To switch to a network MetaMask already knows about use `wallet_switchEthereumChain`.

The following function attempts to switch networks with fallback to adding if unknown.

```
async function switchNetwork(chainIdHex) {  
  try {  
    await window.ethereum.request({
```

```
method: 'wallet_switchEthereumChain',  
params: [{ chainId: chainIdHex }]  
});  
} catch (error) {  
  if (error.code === 4902) {  
    throw new Error('Network not found in wallet');  
  }  
  throw error;  
}  
}
```

Error code 4902 indicates the chain is not added to MetaMask. Your application can catch this and call `wallet_addEthereumChain` to add it.

SECTION 3: WALLETCONNECT PROTOCOL

Understanding WalletConnect

WalletConnect is an open protocol for connecting desktop applications to mobile wallets. It enables users to interact with Web3 applications from any device using their preferred mobile wallet.

The protocol works through relay servers. Your application generates a connection request encoded as a QR code or deep link. The user scans this with their mobile wallet. Both sides establish an encrypted session through the relay allowing them to exchange messages.

WalletConnect v2 introduced major improvements including multi-chain sessions support for multiple parallel sessions and better reliability through the relay network.

WalletConnect v2 Implementation

The following code demonstrates WalletConnect v2 integration using the Web3Modal library which simplifies implementation.

First install the required packages.

```
npm install @web3modal/wagmi wagmi viem @tanstack/react-query
```

The following React component implements WalletConnect with Web3Modal.

```
import { createWeb3Modal, defaultWagmiConfig } from
 '@web3modal/wagmi/react'
import { WagmiConfig } from 'wagmi'
import { mainnet, polygon, arbitrum } from 'wagmi/chains'
import { QueryClient, QueryClientProvider } from
 '@tanstack/react-query'

const projectId = 'YOUR_WALLETCONNECT_PROJECT_ID'

const metadata = {
  name: 'My Crypto App',
  description: 'A blockchain application',
  url: 'https://myapp.com',
  icons: ['https://myapp.com/icon.png']
}

const chains = [mainnet, polygon, arbitrum]

const wagmiConfig = defaultWagmiConfig({
  chains,
  projectId,
  metadata
})
```



```
createWeb3Modal({  
  wagmiConfig,  
  projectId,  
  chains  
})
```

```
const queryClient = new QueryClient()
```

```
function App() {  
  return (  
    <WagmiConfig config={wagmiConfig}>  
    <QueryClientProvider client={queryClient}>  
    <w3m-button />  
    </QueryClientProvider>  
    </WagmiConfig>  
  )  
}
```

You need a project ID from cloud.walletconnect.com. The `w3m-button` component renders a connect button that opens Web3Modal displaying QR codes for mobile wallets and options for installed browser wallets.

Using Connected Wallet with Wagmi

Once connected Wagmi hooks provide access to wallet state and actions.

The following component reads account data and sends transactions.

```
import { useAccount, useBalance, useSendTransaction,  
  useWaitForTransaction } from 'wagmi'  
import { parseEther } from 'viem'  
  
function WalletInterface() {
```

```
const { address, isConnected, chain } = useAccount()
```

```
const { data: balance } = useBalance({  
  address: address  
})
```

```
const {  
  sendTransaction,  
  data: txData,  
  isLoading: isSending  
} = useSendTransaction()
```

```
const { isLoading: isConfirming, isSuccess } =  
useWaitForTransaction({  
  hash: txData?.hash  
})
```

```
const handleSend = () => {  
  sendTransaction({  
    to: '0x742d35Cc6634C0532925a3b844Bc9e7595f8fE1b',  
    value: parseEther('0.01')  
  })  
}
```

```
if (!isConnected) {  
  return <w3m-button />  
}
```

```
return (  
  <div>  
    <p>Connected: {address}</p>  
    <p>Network: {chain?.name}</p>  
    <p>Balance: {balance?.formatted} {balance?.symbol}</p>  
  <p>  
    <button onClick={handleSend} disabled={isSending ||  
isConfirming}>  
      {isSending ? 'Confirm in wallet...' :  
}
```

```

isConfirming ? 'Confirming...' :
'Send 0.01 ETH'}
</button>
{isSuccess && <p>Transaction confirmed! </p> }
</div>
)
}

```

The useAccount hook provides connection state. useBalance fetches the native token balance. useSendTransaction returns a function to initiate transactions and tracks their state. useWaitForTransaction monitors transaction confirmation.

Direct WalletConnect Implementation

For applications not using React here is a direct implementation.

```
import SignClient from '@walletconnect/sign-client'
```

```

async function initializeWalletConnect() {
  const client = await SignClient.init({
    projectId: 'YOUR_PROJECT_ID',
    metadata: {
      name: 'My App',
      description: 'My Application',
      url: 'https://myapp.com',
      icons: ['https://myapp.com/icon.png']
    }
  })
}

```

```

  return client
}

```

```

async function connectWallet(client) {
  const { uri, approval } = await client.connect({

```

```

requiredNamespaces: {
  eip155: {
    methods: [
      'eth_sendTransaction',
      'eth_signTransaction',
      'eth_sign',
      'personal_sign',
      'eth_signTypedData'
    ],
    chains: ['eip155:1', 'eip155:137'],
    events: ['chainChanged', 'accountsChanged']
  }
}
})

```

```

if (uri) {
  console.log('Connect URI:', uri)
}

```

```

const session = await approval()

```

```

return session
}

```

```

async function sendRequest(client, session, chainId, method,
params) {
  const result = await client.request({
    topic: session.topic,
    chainId: `eip155:${chainId}`,
    request: {
      method: method,
      params: params
    }
  })
}

```

```

return result
}

```

The connect function returns a URI that encodes the connection request. Display this as a QR code for users to scan. The approval promise resolves when the user accepts the connection in their wallet.

SECTION 4: COINBASE WALLET SDK

Coinbase Wallet Overview

Coinbase Wallet is distinct from the Coinbase exchange. It is a non-custodial wallet that users control themselves. The Coinbase Wallet SDK allows integration with both the browser extension and mobile wallet.

The SDK provides a seamless experience where users without the extension installed can scan a QR code to connect their mobile wallet. This makes Coinbase Wallet accessible to users on any device.

Installing and Configuring

The following code shows Coinbase Wallet SDK setup.

```
npm install @coinbase/wallet-sdk
```

```
import CoinbaseWalletSDK from '@coinbase/wallet-sdk'
```

```
const sdk = new CoinbaseWalletSDK({  
  appName: 'My Crypto Application',  
  appLogoUrl: 'https://myapp.com/logo.png',  
  darkMode: false  
})
```

```
const provider = sdk.makeWeb3Provider('https://  
mainnet.infura.io/v3/YOUR_KEY', 1)
```

The `makeWeb3Provider` function creates an EIP-1193 compatible provider. The first argument is a fallback RPC URL used when the user has no wallet extension. The second argument is the default chain ID.

Connecting and Using Coinbase Wallet

The provider works identically to MetaMask's injected provider.

```
async function connectCoinbaseWallet(provider) {  
  const accounts = await provider.request({  
    method: 'eth_requestAccounts'  
  })  
  
  return accounts[0]  
}  
  
async function getBalanceCoinbase(provider, address) {  
  const balance = await provider.request({  
    method: 'eth_getBalance',  
    params: [address, 'latest']  
  })  
  
  return balance  
}  
  
async function sendTransactionCoinbase(provider, from, to,  
value) {  
  const txHash = await provider.request({  
    method: 'eth_sendTransaction',  
    params: [{  
      from: from,  
      to: to,  
      value: value
```

```
}]  
})
```

```
return txHash  
}
```

If the user has Coinbase Wallet extension installed the SDK uses it directly. Otherwise it displays a QR code for mobile connection. This happens automatically without code changes.

Handling Multiple Providers

Modern users often have multiple wallet extensions installed. When your application accesses `window.ethereum` you might get any of them.

The following code handles multiple wallet detection.

```
function detectInstalledWallets() {  
  const wallets = []  
  
  if (typeof window.ethereum === 'undefined') {  
    return wallets  
  }  
  
  if (window.ethereum.providers) {  
    for (const provider of window.ethereum.providers) {  
      if (provider.isMetaMask) {  
        wallets.push({ name: 'MetaMask', provider: provider })  
      }  
      if (provider.isCoinbaseWallet) {  
        wallets.push({ name: 'Coinbase Wallet', provider: provider })  
      }  
      if (provider.isRabby) {  
        wallets.push({ name: 'Rabby', provider: provider })  
      }  
    }  
  }  
}
```

```

    }
  } else {
    if (window.ethereum.isMetaMask) {
      wallets.push({ name: 'MetaMask', provider: window.ethereum
    })
    }
    if (window.ethereum.isCoinbaseWallet) {
      wallets.push({ name: 'Coinbase Wallet', provider:
window.ethereum })
    }
  }

  return wallets
}

```

When multiple wallets are installed they may use `window.ethereum.providers` array. Your UI should let users choose which wallet to connect. Store the selected provider and use it for all subsequent requests.

SECTION 5: MULTI-CHAIN SUPPORT

Chain Identification

Each blockchain network has a unique chain ID. Ethereum mainnet is 1. Polygon is 137. Arbitrum is 42161. Your application must track which chain the user is connected to.

The following code retrieves the current chain ID.

```

async function getCurrentChainId() {
  const chainIdHex = await window.ethereum.request({
    method: 'eth_chainId'
  })

  return parseInt(chainIdHex, 16)
}

```



```
}
```

Chain IDs are returned as hexadecimal strings. Convert to integer for comparison.

Chain Configuration Registry

Maintain a registry of supported chains with their configuration.

```
const CHAIN_CONFIG = {  
  1: {  
    name: 'Ethereum',  
    currency: 'ETH',  
    explorer: 'https://etherscan.io',  
    rpcUrl: 'https://mainnet.infura.io/v3/YOUR_KEY'  
  },  
  137: {  
    name: 'Polygon',  
    currency: 'MATIC',  
    explorer: 'https://polygonscan.com',  
    rpcUrl: 'https://polygon-rpc.com'  
  },  
  42161: {  
    name: 'Arbitrum',  
    currency: 'ETH',  
    explorer: 'https://arbiscan.io',  
    rpcUrl: 'https://arb1.arbitrum.io/rpc'  
  },  
  10: {  
    name: 'Optimism',  
    currency: 'ETH',  
    explorer: 'https://optimistic.etherscan.io',  
    rpcUrl: 'https://mainnet.optimism.io'  
  },  
  8453: {
```

```

name: 'Base',
currency: 'ETH',
explorer: 'https://basescan.org',
rpcUrl: 'https://mainnet.base.org'
},
43114: {
name: 'Avalanche',
currency: 'AVAX',
explorer: 'https://snowtrace.io',
rpcUrl: 'https://api.avax.network/ext/bc/C/rpc'
},
56: {
name: 'BNB Chain',
currency: 'BNB',
explorer: 'https://bscscan.com',
rpcUrl: 'https://bsc-dataseed.binance.org'
}
}

```

```

function getChainConfig(chainId) {
  return CHAIN_CONFIG[chainId] || null
}

```

```

function isSupportedChain(chainId) {
  return chainId in CHAIN_CONFIG
}

```

Chain Switching Interface

The following component provides a chain selector.

```

function ChainSelector({ currentChainId, onSwitch }) {
  const supportedChains = Object.entries(CHAIN_CONFIG)

  async function handleSwitch(chainId) {
    const chainIdHex = '0x' + chainId.toString(16)

```

```

try {
  await window.ethereum.request({
    method: 'wallet_switchEthereumChain',
    params: [{ chainId: chainIdHex }]
  })
  onSwitch(chainId)
} catch (error) {
  if (error.code === 4902) {
    await addChain(chainId)
  }
}
}

async function addChain(chainId) {
  const config = CHAIN_CONFIG[chainId]
  const chainIdHex = '0x' + chainId.toString(16)

  await window.ethereum.request({
    method: 'wallet_addEthereumChain',
    params: [{
      chainId: chainIdHex,
      chainName: config.name,
      nativeCurrency: {
        name: config.currency,
        symbol: config.currency,
        decimals: 18
      },
      rpcUrls: [config.rpcUrl],
      blockExplorerUrls: [config.explorer]
    }]
  })
}

return (
  <select
    value = {currentChainId}

```

```

onChange = {(e) => handleSwitch(Number(e.target.value))}
>
{supportedChains.map(([id, config]) => (
  <option key={id} value={id}>{config.name}</option>
))}
</select>
)
}

```

Multi-Chain Contract Addresses

Contracts deployed on different chains have different addresses. Maintain address mappings per chain.

```

const CONTRACT_ADDRESSES = {
  myToken: {
    1: '0x1234567890abcdef1234567890abcdef12345678',
    137: '0xabcdef1234567890abcdef1234567890abcdef12',
    42161: '0x567890abcdef1234567890abcdef1234567890ab'
  },
  myNFT: {
    1: '0xfedcba0987654321fedcba0987654321fedcba09',
    137: '0x0987654321fedcba0987654321fedcba09876543'
  }
}

```

```

function getContractAddress(contractName, chainId) {
  const addresses = CONTRACT_ADDRESSES[contractName]

  if (!addresses) {
    throw new Error(`Unknown contract: ${contractName}`)
  }

  const address = addresses[chainId]

  if (!address) {

```

```
    throw new Error(`Contract ${contractName} not deployed on  
chain ${chainId}`)  
}  
  
return address  
}
```

SECTION 6: TRANSACTION SIGNING IN DEPTH

Transaction Structure

An Ethereum transaction contains several fields each serving a specific purpose.

The from field is the sender address. This is automatically set to the connected account.

The to field is the recipient. For contract calls this is the contract address. For simple transfers this is the receiving wallet.

The value field is the amount of ETH to send in wei. For pure contract calls with no ETH transfer this is zero.

The data field contains the encoded function call for contract interactions. For simple ETH transfers this is empty.

The gas field sets the maximum gas units the transaction can use. If the transaction uses more than this limit it fails but you still pay for the gas used.

The maxFeePerGas and maxPriorityFeePerGas fields are EIP-1559 fee parameters. maxFeePerGas is the maximum total fee per gas unit you will pay. maxPriorityFeePerGas is the tip to validators.

The nonce is a sequential number preventing replay attacks.

Each account has a nonce that increments with every transaction.

Encoding Contract Calls

To call a contract function you must encode the function signature and parameters into the data field. Libraries like ethers.js and viem handle this.

The following example uses viem to encode a contract call.

```
import { encodeFunctionData, parseAbi } from 'viem'

const abi = parseAbi([
  'function transfer(address to, uint256 amount) returns (bool)',
  'function approve(address spender, uint256 amount) returns (bool)',
  'function balanceOf(address account) view returns (uint256)'
])

function encodeTransfer(toAddress, amount) {
  const data = encodeFunctionData({
    abi: abi,
    functionName: 'transfer',
    args: [toAddress, amount]
  })

  return data
}

async function executeTransfer(tokenContract, toAddress, amount) {
  const accounts = await window.ethereum.request({
    method: 'eth_accounts'
  })
}
```

```

const data = encodeTransfer(toAddress, amount)

const txHash = await window.ethereum.request({
  method: 'eth_sendTransaction',
  params: [{
    from: accounts[0],
    to: tokenContract,
    data: data
  }]
})

return txHash
}

```

The `encodeFunctionData` function produces the hexadecimal data string that tells the contract which function to execute and with what parameters.

Gas Estimation

Before sending a transaction you can estimate the gas required. This helps set appropriate gas limits and display expected costs to users.

```

async function estimateGas(transaction) {
  const estimate = await window.ethereum.request({
    method: 'eth_estimateGas',
    params: [transaction]
  })

  return BigInt(estimate)
}

```

```

async function getCurrentGasPrice() {
  const gasPrice = await window.ethereum.request({
    method: 'eth_gasPrice'
  })
}

```

```

    })

    return BigInt(gasPrice)
  }

  async function estimateTransactionCost(transaction) {
    const gasEstimate = await estimateGas(transaction)
    const gasPrice = await getCurrentGasPrice()

    const buffer = gasEstimate * BigInt(120) / BigInt(100)

    const costWei = buffer * gasPrice
    const costEth = Number(costWei) / 1e18

    return {
      gasLimit: buffer.toString(),
      gasPrice: gasPrice.toString(),
      estimatedCostEth: costEth.toFixed(6)
    }
  }
}

```

The buffer adds 20% to the gas estimate. Transaction complexity can vary slightly and underestimating causes failures.

Transaction Receipt Monitoring

After sending a transaction you receive a hash but the transaction is not yet confirmed. Monitor the receipt to track confirmation.

```

async function waitForReceipt(txHash, maxAttempts = 60) {
  for (let i = 0; i < maxAttempts; i++) {
    const receipt = await window.ethereum.request({
      method: 'eth_getTransactionReceipt',
      params: [txHash]
    })
  }
}

```



```
})
```

```
if (receipt !== null) {  
  return {  
    success: receipt.status === '0x1',  
    blockNumber: parseInt(receipt.blockNumber, 16),  
    gasUsed: parseInt(receipt.gasUsed, 16),  
    logs: receipt.logs  
  }  
}
```

```
await new Promise(resolve => setTimeout(resolve, 2000))  
}
```

```
throw new Error('Transaction not confirmed within timeout')  
}
```

The receipt status of 0x1 indicates success. 0x0 indicates the transaction was included in a block but reverted during execution. This can happen if a require statement fails or the contract throws an error.

Speed Up and Cancel Transactions

If a transaction is stuck pending you can replace it by sending a new transaction with the same nonce but higher gas price.

```
async function getTransactionCount(address) {  
  const count = await window.ethereum.request({  
    method: 'eth_getTransactionCount',  
    params: [address, 'latest']  
  })  
  
  return parseInt(count, 16)  
}
```

```

async function speedUpTransaction(originalTx,
newMaxFeePerGas) {
  const accounts = await window.ethereum.request({
    method: 'eth_accounts'
  })

  const nonce = await getPendingNonce(accounts[0])

  const newTx = {
    from: accounts[0],
    to: originalTx.to,
    value: originalTx.value,
    data: originalTx.data,
    nonce: '0x' + nonce.toString(16),
    maxFeePerGas: newMaxFeePerGas
  }

  return await window.ethereum.request({
    method: 'eth_sendTransaction',
    params: [newTx]
  })
}

async function cancelTransaction(newMaxFeePerGas) {
  const accounts = await window.ethereum.request({
    method: 'eth_accounts'
  })

  const nonce = await getPendingNonce(accounts[0])

  const cancelTx = {
    from: accounts[0],
    to: accounts[0],
    value: '0x0',
    nonce: '0x' + nonce.toString(16),
    maxFeePerGas: newMaxFeePerGas
  }

```

```

return await window.ethereum.request({
  method: 'eth_sendTransaction',
  params: [cancelTx]
})
}

```

A cancel is simply sending zero value to yourself with the same nonce. Whichever transaction (original or replacement) gets included first invalidates the other.

SECTION 7: COMPLETE WALLET INTEGRATION PATTERN

React Integration Pattern

The following code presents a complete reusable wallet integration for React applications.

```

import { createContext, useContext, useState, useEffect,
useCallback } from 'react'

```

```

const WalletContext = createContext(null)

```

```

function WalletProvider({ children }) {
  const [account, setAccount] = useState(null)
  const [chainId, setChainId] = useState(null)
  const [isConnecting, setIsConnecting] = useState(false)
  const [error, setError] = useState(null)

```

```

  const isConnected = account !== null

```

```

  const connect = useCallback(async () => {
    if (typeof window.ethereum === 'undefined') {
      setError('No wallet detected')
    }
    return
  }, [])

```

```
setIsConnecting(true)
setError(null)
```

```
try {
  const accounts = await window.ethereum.request({
    method: 'eth_requestAccounts'
  })
```

```
  const chain = await window.ethereum.request({
    method: 'eth_chainId'
  })
```

```
  setAccount(accounts[0])
  setChainId(parseInt(chain, 16))
} catch (err) {
  setError(err.message)
} finally {
  setIsConnecting(false)
}
}, [])
```

```
const disconnect = useCallback(() => {
  setAccount(null)
  setChainId(null)
}, [])
```

```
const switchChain = useCallback(async (newChainId) => {
  const chainIdHex = '0x' + newChainId.toString(16)
```

```
  try {
    await window.ethereum.request({
      method: 'wallet_switchEthereumChain',
      params: [{ chainId: chainIdHex }]
    })
  } catch (err) {
    if (err.code === 4902) {
      setError('Chain not available in wallet')
```

```
}  
throw err  
}  
, [])
```

```
useEffect(() => {  
  if (typeof window.ethereum === 'undefined') return
```

```
  const handleAccountsChanged = (accounts) => {  
    if (accounts.length === 0) {  
      setAccount(null)  
    } else {  
      setAccount(accounts[0])  
    }  
  }  
}
```

```
const handleChainChanged = (chain) => {  
  setChainId(parseInt(chain, 16))  
}
```

```
window.ethereum.on('accountsChanged',  
  handleAccountsChanged)  
window.ethereum.on('chainChanged', handleChainChanged)
```

```
window.ethereum.request({ method: 'eth_accounts' })  
  .then(accounts => {  
    if (accounts.length > 0) {  
      setAccount(accounts[0])  
      window.ethereum.request({ method: 'eth_chainId' })  
        .then(chain => setChainId(parseInt(chain, 16)))  
    }  
  })
```

```
return () => {  
  window.ethereum.removeListener('accountsChanged',  
    handleAccountsChanged)  
  window.ethereum.removeListener('chainChanged',
```

```
handleChainChanged)
}
}, [])
```

```
const value = {
  account,
  chainId,
  isConnected,
  isConnecting,
  error,
  connect,
  disconnect,
  switchChain
}
```

```
return (
  <WalletContext.Provider value={value}>
    {children}
  </WalletContext.Provider>
)
```

```
function useWallet() {
  const context = useContext(WalletContext)
  if (context === null) {
    throw new Error('useWallet must be used within
    WalletProvider')
  }
  return context
}
```

Using the Wallet Context

The following component demonstrates using the wallet context.

```

function WalletButton() {
  const {
    account,
    chainId,
    isConnected,
    isConnecting,
    connect,
    disconnect
  } = useWallet()

  if (isConnecting) {
    return <button disabled> Connecting... </button>
  }

  if (isConnected) {
    const shortAddress = account.slice(0, 6) + '...' +
account.slice(-4)
    const chainName = CHAIN_CONFIG[chainId]?.name ||
'Unknown'

    return (
      <div>
        <span> {shortAddress} on {chainName} </span>
        <button onClick = {disconnect}> Disconnect </button>
      </div>
    )
  }

  return <button onClick = {connect}> Connect Wallet </
button>
}

```

Vanilla JavaScript Pattern

For applications not using React here is a class-based wallet manager.

```

class WalletManager {
  constructor() {
    this.account = null
    this.chainId = null
    this.listeners = {
      accountChanged: [],
      chainChanged: [],
      connected: [],
      disconnected: []
    }

    this.init()
  }

  init() {
    if (typeof window.ethereum === 'undefined') return

    window.ethereum.on('accountsChanged', (accounts) => {
      if (accounts.length === 0) {
        this.account = null
        this.emit('disconnected')
      } else {
        this.account = accounts[0]
        this.emit('accountChanged', this.account)
      }
    })

    window.ethereum.on('chainChanged', (chainId) => {
      this.chainId = parseInt(chainId, 16)
      this.emit('chainChanged', this.chainId)
    })

    this.checkExistingConnection()
  }

  async checkExistingConnection() {

```



```
if (typeof window.ethereum === 'undefined') return
```

```
const accounts = await window.ethereum.request({  
  method: 'eth_accounts'  
})
```

```
if (accounts.length > 0) {  
  this.account = accounts[0]  
  const chainId = await window.ethereum.request({  
    method: 'eth_chainId'  
  })  
  this.chainId = parseInt(chainId, 16)  
  this.emit('connected', this.account)  
}  
}
```

```
async connect() {  
  if (typeof window.ethereum === 'undefined') {  
    throw new Error('No wallet detected')  
  }  
}
```

```
const accounts = await window.ethereum.request({  
  method: 'eth_requestAccounts'  
})
```

```
this.account = accounts[0]
```

```
const chainId = await window.ethereum.request({  
  method: 'eth_chainId'  
})
```

```
this.chainId = parseInt(chainId, 16)
```

```
this.emit('connected', this.account)
```

```
return this.account  
}
```

```
disconnect() {  
  this.account = null  
  this.chainId = null  
  this.emit('disconnected')  
}
```

```
async sendTransaction(to, value, data = '0x') {  
  if (!this.account) {  
    throw new Error('Not connected')  
  }
```

```
  const txHash = await window.ethereum.request({  
    method: 'eth_sendTransaction',  
    params: [{  
      from: this.account,  
      to: to,  
      value: value,  
      data: data  
    }]  
  })
```

```
  return txHash  
}
```

```
async signMessage(message) {  
  if (!this.account) {  
    throw new Error('Not connected')  
  }
```

```
  const signature = await window.ethereum.request({  
    method: 'personal_sign',  
    params: [message, this.account]  
  })
```

```
  return signature  
}
```

```
on(event, callback) {  
  if (this.listeners[event]) {  
    this.listeners[event].push(callback)  
  }  
}
```

```
off(event, callback) {  
  if (this.listeners[event]) {  
    this.listeners[event] = this.listeners[event]  
      .filter(cb => cb !== callback)  
  }  
}
```

```
emit(event, data) {  
  if (this.listeners[event]) {  
    this.listeners[event].forEach(cb => cb(data))  
  }  
}
```

```
get isConnected() {  
  return this.account !== null  
}  
}
```

```
const wallet = new WalletManager()
```

SECTION 8: SECURITY CONSIDERATIONS

Never Trust the Client

The connected wallet address comes from the user's browser. You cannot trust this for authorization of sensitive operations. Always verify signatures on your backend.

The following shows backend signature verification.

```
import { verifyMessage } from 'ethers'

function verifySignature(message, signature, expectedAddress)
{
  const recoveredAddress = verifyMessage(message, signature)
  return recoveredAddress.toLowerCase() == =
  expectedAddress.toLowerCase()
}

function createAuthMessage(address, nonce, timestamp) {
  return `Sign this message to authenticate.

Address: ${address}
Nonce: ${nonce}
Timestamp: ${timestamp}

This signature does not authorize any transactions.`
}
```

Users can modify JavaScript on your page. They can send any address they want. Only a valid signature proves they control the private key.

Phishing Protection

Display clear information about what users are signing. EIP-712 typed data is preferable to raw message signing because wallets display structured data clearly.

Never request signatures for sensitive messages without clear user understanding. A signed message can be used anywhere not just on your site. Make messages domain-specific and include timestamps or nonces.

Transaction Simulation

Before submitting transactions simulate them to check for errors. Services like Tenderly offer transaction simulation APIs.

```
async function simulateTransaction(tx) {  
  const result = await window.ethereum.request({  
    method: 'eth_call',  
    params: [tx, 'latest']  
  })  
  
  return result  
}
```

The `eth_call` method executes the transaction logic without submitting it to the network. If it reverts you can catch the error before the user pays gas for a failed transaction.

Rate Limiting and Validation

Validate all inputs before creating transactions. Check address formats amount ranges and data integrity.

```
function isValidAddress(address) {  
  return /^0x[a-fA-F0-9]{40}$/.test(address)  
}  
  
function isValidAmount(amount, decimals = 18) {  
  if (typeof amount !== 'string' && typeof amount !==  
  'number') {  
    return false  
  }  
  
  const value = Number(amount)  
  return !isNaN(value) && value > 0 && isFinite(value)  
}
```

```
function sanitizeHex(hex) {
  if (typeof hex !== 'string') return null
  if (!hex.startsWith('0x')) return null
  if (!/^0x[a-fA-F0-9]*$/.test(hex)) return null
  return hex.toLowerCase()
}
```

SECTION 9: TESTING WALLET INTEGRATION

Mock Provider for Testing

Create a mock provider that simulates wallet behavior for testing.

```
class MockProvider {
  constructor(config = {}) {
    this.accounts = config.accounts || [
      '0x742d35Cc6634C0532925a3b844Bc9e7595f8fE1b'
    ]
    this.chainId = config.chainId || 1
    this.balance = config.balance || '0x56bc75e2d63100000'
    this.listeners = {}
  }

  async request({ method, params }) {
    switch (method) {
      case 'eth_requestAccounts':
      case 'eth_accounts':
        return this.accounts

      case 'eth_chainId':
        return '0x' + this.chainId.toString(16)

      case 'eth_getBalance':
        return this.balance
    }
  }
}
```

```
case 'eth_sendTransaction':  
return '0x' + '1'.repeat(64)
```

```
case 'personal_sign':  
return '0x' + 'a'.repeat(130)
```

```
case 'wallet_switchEthereumChain':  
this.chainId = parseInt(params[0].chainId, 16)  
this.emit('chainChanged', params[0].chainId)  
return null
```

```
default:  
throw new Error(`Unsupported method: ${method}`)  
}  
}
```

```
on(event, callback) {  
if (!this.listeners[event]) {  
this.listeners[event] = []  
}  
this.listeners[event].push(callback)  
}
```

```
removeListener(event, callback) {  
if (this.listeners[event]) {  
this.listeners[event] = this.listeners[event]  
.filter(cb => cb !== callback)  
}  
}
```

```
emit(event, data) {  
if (this.listeners[event]) {  
this.listeners[event].forEach(cb => cb(data))  
}  
}
```

```
simulateAccountChange(newAccounts) {
```

```

this.accounts = newAccounts
this.emit('accountsChanged', newAccounts)
}

simulateChainChange(newChainId) {
this.chainId = newChainId
this.emit('chainChanged', '0x' + newChainId.toString(16))
}

simulateDisconnect() {
this.accounts = []
this.emit('accountsChanged', [])
}
}

```

Writing Integration Tests

The following tests verify wallet integration behavior.

```

describe('Wallet Integration', () => {
  let mockProvider

  beforeEach(() => {
    mockProvider = new MockProvider({
      accounts:
['0x742d35Cc6634C0532925a3b844Bc9e7595f8fE1b'],
      chainId: 1
    })
    window.ethereum = mockProvider
  })

  test('connects wallet and returns address', async () => {
    const address = await connectWallet()
    expect(address).toBe('0x742d35Cc6634C0532925a3b844Bc9e7595f8fE1b')
  })
}

```



```
test('handles account change event', async () => {  
  let receivedAccount = null
```

```
  setupWalletListeners(  
    (account) => { receivedAccount = account },  
    () => {}  
  )
```

```
  mockProvider.simulateAccountChange([  
    '0x1234567890123456789012345678901234567890'  
  ])
```

```
  expect(receivedAccount).toBe(  
    '0x1234567890123456789012345678901234567890'  
  )  
})
```

```
test('handles disconnect', async () => {  
  let disconnected = false
```

```
  setupWalletListeners(  
    (account) => { disconnected = account === null },  
    () => {}  
  )
```

```
  mockProvider.simulateDisconnect()
```

```
  expect(disconnected).toBe(true)  
})
```

```
test('switches chain correctly', async () => {  
  await switchNetwork('0x89')  
  expect(mockProvider.chainId).toBe(137)  
})  
})
```

SECTION 10: TROUBLESHOOTING COMMON ISSUES

Wallet Not Detected

Users report no wallet detected even with MetaMask installed.

The issue may be that the page loads before the wallet injects `window.ethereum`. Add a delay or check on user action rather than page load.

```
async function connectWithRetry(maxAttempts = 5) {  
  for (let i = 0; i < maxAttempts; i++) {  
    if (typeof window.ethereum !== 'undefined') {  
      return await connectWallet()  
    }  
    await new Promise(resolve => setTimeout(resolve, 500))  
  }  
  throw new Error('Wallet not detected after multiple  
attempts')  
}
```

Transaction Stuck Pending

A transaction shows pending indefinitely.

The likely cause is a low gas price during network congestion. The solution is to speed up or cancel the transaction using the same nonce with higher gas.

User Rejected Request

The `eth_requestAccounts` call throws error code 4001.

This means the user clicked reject in the wallet popup. Handle this gracefully.

```

async function connectWithRejectionHandling() {
  try {
    return await connectWallet()
  } catch (error) {
    if (error.code === 4001) {
      return { rejected: true, message: 'Connection rejected by user' }
    }
    throw error
  }
}

```

Wrong Network

Users attempt transactions on the wrong network.

Check the chain ID before every transaction and prompt to switch if needed.

```

async function ensureCorrectNetwork(requiredChainId) {
  const currentChainId = await getCurrentChainId()

  if (currentChainId !== requiredChainId) {
    await switchNetwork('0x' + requiredChainId.toString(16))
  }
}

```

CHAPTER SUMMARY

This chapter covered the complete landscape of wallet integration from understanding different wallet types to implementing production-ready connection flows.

We explored browser extension wallets like MetaMask and how they inject providers into web pages. The EIP-1193

standard ensures compatibility across different wallet implementations.

WalletConnect protocol enables mobile wallet connections through QR codes and relay servers. Coinbase Wallet SDK provides another integration path with automatic fallback to mobile when no extension is installed.

Multi-chain support requires careful chain ID tracking network switching and maintaining contract address registries per chain.

Transaction signing involves encoding function calls estimating gas monitoring receipts and handling stuck transactions.

Security considerations include never trusting client-provided addresses verifying signatures on the backend and protecting users from phishing through clear signing messages.

Testing wallet integration requires mock providers that simulate wallet behavior without real connections.

With wallet integration complete users can connect to your application sign messages and submit transactions. The next chapter covers backend services that listen for blockchain events process transactions and maintain synchronized state.

CHAPTER 9: BACKEND SERVICES FOR BLOCKCHAIN

Frontend wallet integration lets users interact with your application but most production blockchain applications require backend services. These services monitor blockchain events index historical data process transactions securely and provide APIs that aggregate on-chain and off-chain information. This chapter covers building robust backend infrastructure for blockchain applications.

SECTION 1: WEB3 LIBRARIES OVERVIEW

Why Use Web3 Libraries

Every blockchain interaction ultimately happens through JSON-RPC calls to nodes. You could make raw HTTP requests but Web3 libraries provide type safety, ABI encoding, event parsing, and connection management that would take thousands of lines to implement yourself.

The major libraries for EVM chains are ethers.js for JavaScript and web3.py for Python. For Solana there is solana-py for Python and solana-web3.js for JavaScript. Each library handles the low-level protocol details letting you focus on application logic.

Ethers.js Setup and Providers

Ethers.js is the most popular JavaScript library for Ethereum interaction. Version 6 is current with significant improvements over version 5.

Install ethers.js in your Node.js project.

```
npm install ethers
```

The provider is your connection to the blockchain. Different provider types connect to different node sources.

```
import { ethers } from 'ethers'
```

```
const infuraProvider = new ethers.JsonRpcProvider(  
  'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'  
)
```

```
const alchemyProvider = new ethers.JsonRpcProvider(  
  'https://eth-mainnet.alchemyapi.io/v2/your-api-key')
```

```

'https://eth-mainnet.g.alchemy.com/v2/YOUR_API_KEY'
)

const localProvider = new ethers.JsonRpcProvider(
  'http://localhost:8545'
)

async function getBlockNumber(provider) {
  const blockNumber = await provider.getBlockNumber()
  return blockNumber
}

async function getBalance(provider, address) {
  const balance = await provider.getBalance(address)
  const balanceInEther = ethers.formatEther(balance)
  return balanceInEther
}

```

The `JsonRpcProvider` connects to any Ethereum node via HTTP. For production use multiple providers with failover to ensure reliability.

Ethers.js Signers and Wallets

A signer is an object that can sign transactions. In backend services you typically use a `Wallet` instance initialized with a private key.

```

import { ethers } from 'ethers'

const provider = new ethers.JsonRpcProvider(
  'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'
)

const wallet = new ethers.Wallet(
  process.env.PRIVATE_KEY,

```

```
provider
)
```

```
async function sendEther(toAddress, amountEther) {
  const tx = await wallet.sendTransaction({
    to: toAddress,
    value: ethers.parseEther(amountEther)
  })
```

```
  const receipt = await tx.wait()
```

```
  return {
    hash: tx.hash,
    blockNumber: receipt.blockNumber,
    gasUsed: receipt.gasUsed.toString()
  }
}
```

Never hardcode private keys in source code. Use environment variables or a secrets manager. The wallet object combines signing capability with provider connectivity.

Contract Interaction with Ethers.js

To interact with smart contracts you need the contract address and ABI.

```
import { ethers } from 'ethers'
```

```
const ERC20_ABI = [
  'function name() view returns (string)',
  'function symbol() view returns (string)',
  'function decimals() view returns (uint8)',
  'function totalSupply() view returns (uint256)',
  'function balanceOf(address account) view returns (uint256)',
  'function transfer(address to, uint256 amount) returns (bool)',
```

```
'function approve(address spender, uint256 amount) returns
(bool)',
'function allowance(address owner, address spender) view
returns (uint256)',
'event Transfer(address indexed from, address indexed to,
uint256 value)',
'event Approval(address indexed owner, address indexed
spender, uint256 value)'
]
```

```
const provider = new ethers.JsonRpcProvider(
  'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'
)
```

```
const usdcAddress =
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
```

```
const usdcContract = new ethers.Contract(
  usdcAddress,
  ERC20_ABI,
  provider
)
```

```
async function getTokenInfo() {
  const name = await usdcContract.name()
  const symbol = await usdcContract.symbol()
  const decimals = await usdcContract.decimals()
  const totalSupply = await usdcContract.totalSupply()

  return {
    name: name,
    symbol: symbol,
    decimals: Number(decimals),
    totalSupply: ethers.formatUnits(totalSupply, decimals)
  }
}
```



```
async function getTokenBalance(address) {  
  const balance = await usdcContract.balanceOf(address)  
  const decimals = await usdcContract.decimals()  
  return ethers.formatUnits(balance, decimals)  
}
```

Ethers.js uses human-readable ABI format. For complex contracts you can import the full JSON ABI from compilation artifacts.

Web3.py Setup

For Python backends web3.py provides comprehensive Ethereum support.

```
pip install web3
```

```
from web3 import Web3
```

```
infura_url = 'https://mainnet.infura.io/v3/  
YOUR_PROJECT_ID'  
w3 = Web3(Web3.HTTPProvider(infura_url))
```

```
def check_connection():  
    return w3.is_connected()
```

```
def get_block_number():  
    return w3.eth.block_number
```

```
def get_balance(address):  
    balance_wei = w3.eth.get_balance(address)  
    balance_eth = w3.from_wei(balance_wei, 'ether')  
    return balance_eth
```

```
def get_transaction(tx_hash):  
    tx = w3.eth.get_transaction(tx_hash)
```

```

return {
'from': tx['from'],
'to': tx['to'],
'value': w3.from_wei(tx['value'], 'ether'),
'gas': tx['gas'],
'block_number': tx['blockNumber']}
}

```

Contract Interaction with Web3.py

Contract interaction in web3.py requires the checksum address and ABI.

```

from web3 import Web3

```

```

w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/v3/YOUR_PROJECT_ID'))

```

```

ERC20_ABI = [
{
"constant": True,
"inputs": [],
"name": "name",
"outputs": [{"name": "", "type": "string"}],
"type": "function"
},
{
"constant": True,
"inputs": [],
"name": "symbol",
"outputs": [{"name": "", "type": "string"}],
"type": "function"
},
{
"constant": True,
"inputs": [],

```

```

"name": "decimals",
"outputs": [{"name": "", "type": "uint8"}],
"type": "function"
},
{
"constant": True,
"inputs": [{"name": "account", "type": "address"}],
"name": "balanceOf",
"outputs": [{"name": "", "type": "uint256"}],
"type": "function"
}
]

```

```

usdc_address = Web3.to_checksum_address(
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
)

```

```

usdc_contract = w3.eth.contract(address=usdc_address,
abi=ERC20_ABI)

```

```

def get_token_info():
    name = usdc_contract.functions.name().call()
    symbol = usdc_contract.functions.symbol().call()
    decimals = usdc_contract.functions.decimals().call()

```

```

    return {
        'name': name,
        'symbol': symbol,
        'decimals': decimals
    }

```

```

def get_token_balance(wallet_address):
    checksum_address =
Web3.to_checksum_address(wallet_address)
    balance =
usdc_contract.functions.balanceOf(checksum_address).call()
    decimals = usdc_contract.functions.decimals().call()

```

return balance / (10 ** decimals)

Sending Transactions with Web3.py

Sending transactions requires a private key and proper nonce management.

```
from web3 import Web3
import os
```

```
w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/
v3/YOUR_PROJECT_ID'))
```

```
private_key = os.environ.get('PRIVATE_KEY')
account = w3.eth.account.from_key(private_key)
```

```
def send_ether(to_address, amount_ether):
    nonce = w3.eth.get_transaction_count(account.address)
```

```
    tx = {
        'nonce': nonce,
        'to': Web3.to_checksum_address(to_address),
        'value': w3.to_wei(amount_ether, 'ether'),
        'gas': 21000,
        'maxFeePerGas': w3.eth.gas_price * 2,
        'maxPriorityFeePerGas': w3.to_wei(2, 'gwei'),
        'chainId': 1
    }
```

```
    signed_tx = w3.eth.account.sign_transaction(tx, private_key)
    tx_hash =
    w3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

```
    receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
```

```
    return {
```

```
'hash': tx_hash.hex(),  
'block_number': receipt['blockNumber'],  
'gas_used': receipt['gasUsed'],  
'status': receipt['status']  
}
```

```
def call_contract_function(contract, function_name, args):  
    nonce = w3.eth.get_transaction_count(account.address)
```

```
    function = getattr(contract.functions, function_name)
```

```
    tx = function(*args).build_transaction({  
        'from': account.address,  
        'nonce': nonce,  
        'maxFeePerGas': w3.eth.gas_price * 2,  
        'maxPriorityFeePerGas': w3.to_wei(2, 'gwei')  
    })
```

```
    signed_tx = w3.eth.account.sign_transaction(tx, private_key)  
    tx_hash =  
    w3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

```
    receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
```

```
    return receipt
```

SECTION 2: RPC CALLS AND METHODS

Standard Ethereum JSON-RPC Methods

All Ethereum nodes implement the JSON-RPC API specification. Understanding these methods helps debug issues and optimize queries.

Reading state methods are free and do not require signing.

`eth_blockNumber` returns the current block number.

`eth_getBalance` returns the ETH balance of an address.
`eth_getTransactionCount` returns the number of transactions sent from an address.
`eth_getCode` returns the bytecode at an address (empty for EOAs).
`eth_call` executes a read-only contract call.
`eth_getTransactionByHash` returns transaction details.
`eth_getTransactionReceipt` returns the receipt of a mined transaction.
`eth_getLogs` returns event logs matching a filter.

State-changing methods require signed transactions.

`eth_sendRawTransaction` broadcasts a signed transaction.
`eth_sendTransaction` requests the node to sign and broadcast (not available on remote nodes).

Making Raw RPC Calls

Sometimes you need to make RPC calls not abstracted by your library.

```
import { ethers } from 'ethers'
```

```
const provider = new ethers.JsonRpcProvider(  
  'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'  
)
```

```
async function rawRpcCall(method, params) {  
  const result = await provider.send(method, params)  
  return result  
}
```

```
async function getStorageAt(contractAddress, slot) {  
  const storage = await rawRpcCall('eth_getStorageAt', [  
    contractAddress,
```

```

ethers.toQuantity(slot),
'latest'
])
return storage
}

```

```

async function traceTransaction(txHash) {
  const trace = await rawRpcCall('debug_traceTransaction', [
    txHash,
    { tracer: 'callTracer' }
  ])
  return trace
}

```

Debug and trace methods are only available on archive nodes with debug APIs enabled. Regular nodes do not support these.

Batch Requests

To reduce latency and API calls batch multiple requests together.

```

async function batchGetBalances(provider, addresses) {
  const batch = addresses.map((address, index) => ({
    method: 'eth_getBalance',
    params: [address, 'latest'],
    id: index,
    jsonrpc: '2.0'
  })))

  const response = await fetch(provider.connection.url, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(batch)
  })
}

```

```

const results = await response.json()

return results.map((result, index) => ({
  address: addresses[index],
  balance: ethers.formatEther(BigInt(result.result))
}))
}

```

Providers often have batch limits (typically 100-1000 requests per batch). Split larger queries appropriately.

Multicall for Contract Reads

Multicall contracts aggregate multiple contract reads into a single RPC call. This dramatically reduces latency and API usage.

```

import { ethers } from 'ethers'

const MULTICALL_ABI = [
  'function aggregate(tuple(address target, bytes callData)[] calls) view returns (uint256 blockNumber, bytes[] returnData)',
  'function tryAggregate(bool requireSuccess, tuple(address target, bytes callData)[] calls) view returns (tuple(bool success, bytes returnData)[] returnData)'
]

const MULTICALL_ADDRESS =
'0xcA11bde05977b3631167028862bE2a173976CA11'

class MulticallHelper {
  constructor(provider) {
    this.provider = provider
    this.contract = new ethers.Contract(
      MULTICALL_ADDRESS,

```



```
MULTICALL_ABI,  
provider  
)  
}
```

```
async aggregate(calls) {  
  const callData = calls.map(call => ({  
    target: call.target,  
    callData: call.interface.encodeFunctionData(  
      call.function,  
      call.args || []  
    )  
  }))
```

```
  const [blockNumber, returnData] = await  
  this.contract.aggregate(callData)
```

```
  return calls.map((call, index) => {  
    const decoded = call.interface.decodeFunctionResult(  
      call.function,  
      returnData[index]  
    )  
    return decoded.length === 1 ? decoded[0] : decoded  
  })  
}  
}
```

```
async function getMultipleTokenBalances(multicall,  
tokenAddresses, walletAddress) {  
  const erc20Interface = new ethers.Interface([  
    'function balanceOf(address) view returns (uint256)'  
  ])
```

```
  const calls = tokenAddresses.map(tokenAddress => ({  
    target: tokenAddress,  
    interface: erc20Interface,  
    function: 'balanceOf',
```

```
args: [walletAddress]  
)))
```

```
const balances = await multicall.aggregate(calls)
```

```
return tokenAddresses.map((address, index) => ({  
  token: address,  
  balance: balances[index].toString()  
}))  
}
```

SECTION 3: EVENT LISTENING

Understanding Blockchain Events

When smart contracts emit events those events are recorded in transaction logs. Events provide an efficient way to track contract activity without polling state changes.

Events are defined in Solidity with the event keyword and emitted with emit. Each event can have up to three indexed parameters that can be filtered efficiently plus additional non-indexed data.

Event logs contain the contract address, topics array (event signature and indexed parameters), data (non-indexed parameters), block number, transaction hash, and log index.

Subscribing to Events with Ethers.js

Ethers.js provides convenient event filtering and subscription.

```
import { ethers } from 'ethers'
```

```
const provider = new ethers.WebSocketProvider(  
  'wss://mainnet.infura.io/ws/v3/YOUR_PROJECT_ID'
```

)

```
const ERC20_ABI = [  
  'event Transfer(address indexed from, address indexed to,  
    uint256 value)'  
]
```

```
const usdcContract = new ethers.Contract(  
  '0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48',  
  ERC20_ABI,  
  provider  
)
```

```
function subscribeToTransfers(callback) {  
  usdcContract.on('Transfer', (from, to, value, event) => {  
    callback({  
      from: from,  
      to: to,  
      value: ethers.formatUnits(value, 6),  
      blockNumber: event.log.blockNumber,  
      transactionHash: event.log.transactionHash  
    })  
  })  
}
```

```
function subscribeToTransfersToAddress(address, callback) {  
  const filter = usdcContract.filters.Transfer(null, address)  
  
  usdcContract.on(filter, (from, to, value, event) => {  
    callback({  
      from: from,  
      to: to,  
      value: ethers.formatUnits(value, 6),  
      transactionHash: event.log.transactionHash  
    })  
  })  
}
```

```
function unsubscribe() {  
  usdcContract.removeAllListeners('Transfer')  
}
```

WebSocket providers are necessary for real-time subscriptions. HTTP providers require polling which adds latency.

Querying Historical Events

To fetch past events use `queryFilter` with a block range.

```
async function getHistoricalTransfers(contract, fromBlock,  
toBlock) {  
  const filter = contract.filters.Transfer()  
  
  const events = await contract.queryFilter(filter, fromBlock,  
toBlock)  
  
  return events.map(event => ({  
    from: event.args[0],  
    to: event.args[1],  
    value: event.args[2].toString(),  
    blockNumber: event.blockNumber,  
    transactionHash: event.transactionHash,  
    logIndex: event.index  
  })))  
}
```

```
async function getTransfersWithPagination(contract,  
startBlock, endBlock, batchSize = 2000) {  
  const allEvents = []  
  
  let currentBlock = startBlock  
  
  while (currentBlock <= endBlock) {
```

```

const batchEnd = Math.min(currentBlock + batchSize - 1,
endBlock)

const events = await getHistoricalTransfers(
contract,
currentBlock,
batchEnd
)

allEvents.push(...events)
currentBlock = batchEnd + 1

await new Promise(resolve => setTimeout(resolve, 100))
}

return allEvents
}

```

Most providers limit event queries to a few thousand blocks. Paginate through larger ranges to avoid timeouts.

Event Listening with Web3.py

Web3.py provides event filtering for Python backends.

```

from web3 import Web3
import asyncio

```

```

w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/
v3/YOUR_PROJECT_ID'))

```

```

TRANSFER_EVENT_ABI = {
    "anonymous": False,
    "inputs": [
        {"indexed": True, "name": "from", "type": "address"},
        {"indexed": True, "name": "to", "type": "address"},

```

```
{ "indexed": False, "name": "value", "type": "uint256" }  
],  
"name": "Transfer",  
"type": "event"  
}
```

```
def get_transfer_events(contract_address, from_block, to_block):  
    contract = w3.eth.contract(  
        address=Web3.to_checksum_address(contract_address),  
        abi=[TRANSFER_EVENT_ABI]  
    )
```

```
    event_filter = contract.events.Transfer.create_filter(  
        fromBlock=from_block,  
        toBlock=to_block  
    )
```

```
    events = event_filter.get_all_entries()
```

```
    return [{  
        'from': event['args']['from'],  
        'to': event['args']['to'],  
        'value': event['args']['value'],  
        'block_number': event['blockNumber'],  
        'transaction_hash': event['transactionHash'].hex()  
    } for event in events]
```

```
def poll_new_events(contract_address, callback,  
    poll_interval=2):  
    contract = w3.eth.contract(  
        address=Web3.to_checksum_address(contract_address),  
        abi=[TRANSFER_EVENT_ABI]  
    )
```

```
    event_filter =  
    contract.events.Transfer.create_filter(fromBlock='latest')
```

```

while True:
    new_events = event_filter.get_new_entries()

    for event in new_events:
        callback({
            'from': event['args']['from'],
            'to': event['args']['to'],
            'value': event['args']['value'],
            'transaction_hash': event['transactionHash'].hex()
        })

    time.sleep(poll_interval)

```

Robust Event Listener Service

Production event listeners need reconnection logic checkpoint management and error handling.

```

import { ethers } from 'ethers'

class EventListenerService {
    constructor(config) {
        this.wsUrl = config.wsUrl
        this.contracts = config.contracts
        this.onEvent = config.onEvent
        this.onError = config.onError || console.error
        this.checkpointStore = config.checkpointStore
        this.provider = null
        this.isRunning = false
        this.reconnectDelay = 1000
    }

    async start() {
        this.isRunning = true
        await this.connect()
    }
}

```

```
async stop() {  
  this.isRunning = false  
  if (this.provider) {  
    this.provider.removeAllListeners()  
    await this.provider.destroy()  
  }  
}
```

```
async connect() {  
  try {  
    this.provider = new ethers.WebSocketProvider(this.wsUrl)  
  
    this.provider.on('error', (error) => {  
      this.onError(error)  
      this.reconnect()  
    })  
  }
```

```
    const websocket = this.provider.websocket  
    websocket.on('close', () => {  
      if (this.isRunning) {  
        this.reconnect()  
      }  
    })  
  }
```

```
  await this.setupListeners()  
  await this.catchUp()
```

```
  this.reconnectDelay = 1000
```

```
  } catch (error) {  
    this.onError(error)  
    this.reconnect()  
  }  
}
```

```
async reconnect() {
```



```
if (!this.isRunning) return
```

```
await new Promise(resolve => setTimeout(resolve,  
this.reconnectDelay))
```

```
this.reconnectDelay = Math.min(this.reconnectDelay * 2,  
30000)
```

```
await this.connect()  
}
```

```
async setupListeners() {  
  for (const contractConfig of this.contracts) {  
    const contract = new ethers.Contract(  
      contractConfig.address,  
      contractConfig.abi,  
      this.provider  
    )
```

```
    for (const eventName of contractConfig.events) {  
      contract.on(eventName, async (...args) => {  
        const event = args[args.length - 1]
```

```
        await this.onEvent({  
          contract: contractConfig.address,  
          event: eventName,  
          args: args.slice(0, -1),  
          blockNumber: event.log.blockNumber,  
          transactionHash: event.log.transactionHash,  
          logIndex: event.log.index  
        })
```

```
        await this.checkpointStore.save(  
          contractConfig.address,  
          event.log.blockNumber  
        )  
      })
```

```
}  
}  
}
```

```
async catchUp() {  
  const currentBlock = await this.provider.getBlockNumber()
```

```
  for (const contractConfig of this.contracts) {  
    const lastProcessed = await this.checkpointStore.get(  
      contractConfig.address  
    )
```

```
    if (lastProcessed && lastProcessed < currentBlock) {  
      const contract = new ethers.Contract(  
        contractConfig.address,  
        contractConfig.abi,  
        this.provider  
      )
```

```
      for (const eventName of contractConfig.events) {  
        const filter = contract.filters[eventName]()  
        const events = await contract.queryFilter(  
          filter,  
          lastProcessed + 1,  
          currentBlock  
        )
```

```
        for (const event of events) {  
          await this.onEvent({  
            contract: contractConfig.address,  
            event: eventName,  
            args: event.args,  
            blockNumber: event.blockNumber,  
            transactionHash: event.transactionHash,  
            logIndex: event.index  
          })  
        }  
      }
```

```

}

await this.checkpointStore.save(
  contractConfig.address,
  currentBlock
)
}
}
}
}
}
}

```

SECTION 4: INDEXING BLOCKCHAIN DATA

Why Index Data

Querying the blockchain directly is slow. Want to know a user's complete transaction history? You would need to scan every block checking every transaction. For addresses that have existed since genesis this could mean billions of comparisons.

Indexing extracts relevant data from the blockchain and stores it in a queryable database. Your API queries the index instead of the blockchain enabling fast complex queries.

Simple Indexer Architecture

A basic indexer follows this pattern: fetch blocks, parse transactions and events, store in database, track progress.

```

import { ethers } from 'ethers'

class SimpleIndexer {
  constructor(config) {
    this.provider = new ethers.JsonRpcProvider(config.rpcUrl)
  }
}

```

```

this.database = config.database
this.contractAddress = config.contractAddress
this.abi = config.abi
this.contract = new ethers.Contract(
this.contractAddress,
this.abi,
this.provider
)
this.batchSize = config.batchSize || 100
}

```

```

async getLastIndexedBlock() {
const result = await this.database.query(
'SELECT MAX(block_number) as last_block FROM
indexed_blocks WHERE contract = $1',
[this.contractAddress]
)
return result.rows[0]?.last_block || 0
}

```

```

async indexRange(startBlock, endBlock) {
const filter = this.contract.filters.Transfer()

```

```

for (let from = startBlock; from <= endBlock; from +=
this.batchSize) {
const to = Math.min(from + this.batchSize - 1, endBlock)

```

```

const events = await this.contract.queryFilter(filter, from, to)

```

```

await this.database.transaction(async (client) => {
for (const event of events) {
await client.query(
`INSERT INTO transfers
(contract, block_number, tx_hash, log_index, from_address,
to_address, value)
VALUES ($1, $2, $3, $4, $5, $6, $7)
ON CONFLICT (contract, tx_hash, log_index) DO NOTHING`,

```

```
[
  this.contractAddress,
  event.blockNumber,
  event.transactionHash,
  event.index,
  event.args[0],
  event.args[1],
  event.args[2].toString()
]
)
}
```

```
await client.query(
`INSERT INTO indexed_blocks (contract, block_number,
indexed_at)
VALUES ($1, $2, NOW())
ON CONFLICT (contract, block_number) DO NOTHING`,
[this.contractAddress, to]
)
})
```

```
console.log(`Indexed blocks ${from} to ${to}`)
}
}
```

```
async run() {
  const lastIndexed = await this.getLastIndexedBlock()
  const currentBlock = await this.provider.getBlockNumber()

  if (lastIndexed < currentBlock) {
    await this.indexRange(lastIndexed + 1, currentBlock)
  }
}
}
```

Database Schema for Blockchain Data

Design schemas that support common query patterns.

```
CREATE TABLE blocks (  
  number BIGINT PRIMARY KEY,  
  hash VARCHAR(66) NOT NULL,  
  parent_hash VARCHAR(66) NOT NULL,  
  timestamp TIMESTAMP NOT NULL,  
  gas_used BIGINT NOT NULL,  
  gas_limit BIGINT NOT NULL,  
  base_fee_per_gas BIGINT,  
  transaction_count INT NOT NULL,  
  indexed_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE INDEX idx_blocks_timestamp ON blocks(timestamp);  
CREATE INDEX idx_blocks_hash ON blocks(hash);
```

```
CREATE TABLE transactions (  
  hash VARCHAR(66) PRIMARY KEY,  
  block_number BIGINT NOT NULL REFERENCES  
  blocks(number),  
  transaction_index INT NOT NULL,  
  from_address VARCHAR(42) NOT NULL,  
  to_address VARCHAR(42),  
  value NUMERIC(78, 0) NOT NULL,  
  gas_price BIGINT NOT NULL,  
  gas_used BIGINT,  
  input_data TEXT,  
  status SMALLINT,  
  indexed_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE INDEX idx_transactions_block ON  
transactions(block_number);  
CREATE INDEX idx_transactions_from ON  
transactions(from_address);
```

```
CREATE INDEX idx_transactions_to ON  
transactions(to_address);  
CREATE INDEX idx_transactions_from_block ON  
transactions(from_address, block_number DESC);
```

```
CREATE TABLE token_transfers (  
  id SERIAL PRIMARY KEY,  
  contract_address VARCHAR(42) NOT NULL,  
  block_number BIGINT NOT NULL,  
  transaction_hash VARCHAR(66) NOT NULL,  
  log_index INT NOT NULL,  
  from_address VARCHAR(42) NOT NULL,  
  to_address VARCHAR(42) NOT NULL,  
  value NUMERIC(78, 0) NOT NULL,  
  indexed_at TIMESTAMP DEFAULT NOW(),  
  UNIQUE(transaction_hash, log_index)  
);
```

```
CREATE INDEX idx_token_transfers_contract ON  
token_transfers(contract_address);  
CREATE INDEX idx_token_transfers_from ON  
token_transfers(from_address);  
CREATE INDEX idx_token_transfers_to ON  
token_transfers(to_address);  
CREATE INDEX idx_token_transfers_block ON  
token_transfers(block_number);
```

```
CREATE TABLE token_balances (  
  address VARCHAR(42) NOT NULL,  
  contract_address VARCHAR(42) NOT NULL,  
  balance NUMERIC(78, 0) NOT NULL,  
  last_updated_block BIGINT NOT NULL,  
  PRIMARY KEY (address, contract_address)  
);
```

```
CREATE INDEX idx_token_balances_contract ON  
token_balances(contract_address);
```

Maintaining Computed State

Some queries require computed state like current token balances. Maintain these as events are processed.

```
class BalanceIndexer {  
    constructor(database) {  
        this.database = database  
    }  
  
    async processTransfer(event) {  
        const { contract_address, from_address, to_address, value,  
            block_number } = event  
  
        await this.database.transaction(async (client) => {  
            await client.query(  
                `INSERT INTO token_transfers  
                (contract_address, block_number, transaction_hash, log_index,  
                from_address, to_address, value)  
                VALUES ($1, $2, $3, $4, $5, $6, $7)  
                ON CONFLICT (transaction_hash, log_index) DO NOTHING`,  
                [  
                    contract_address,  
                    block_number,  
                    event.transaction_hash,  
                    event.log_index,  
                    from_address,  
                    to_address,  
                    value  
                ]  
            )  
  
            if (from_address !==  
'0x0000000000000000000000000000000000000000') {  
                await client.query(  

```



```

`INSERT INTO token_balances (address, contract_address,
balance, last_updated_block)
VALUES ($1, $2, -$3, $4)
ON CONFLICT (address, contract_address) DO UPDATE
SET balance = token_balances.balance - EXCLUDED.balance,
last_updated_block = EXCLUDED.last_updated_block`,
[from_address, contract_address, value, block_number]
)
}

if (to_address != =
'0x0000000000000000000000000000000000000000') {
  await client.query(
    `INSERT INTO token_balances (address, contract_address,
balance, last_updated_block)
VALUES ($1, $2, $3, $4)
ON CONFLICT (address, contract_address) DO UPDATE
SET balance = token_balances.balance +
EXCLUDED.balance,
last_updated_block = EXCLUDED.last_updated_block`,
[to_address, contract_address, value, block_number]
)
}
})
}
}

```

Handling Chain Reorganizations

Blockchains can reorganize when competing blocks exist. A block you indexed might be replaced by a different block at the same height. Your indexer must handle reorgs.

```

class ReorgAwareIndexer {
  constructor(config) {
    this.provider = config.provider
  }
}

```

```
this.database = config.database
this.confirmationBlocks = config.confirmationBlocks || 12
}
```

```
async checkForReorg() {
  const currentBlock = await this.provider.getBlockNumber()
  const safeBlock = currentBlock - this.confirmationBlocks
```

```
  const recentBlocks = await this.database.query(
    'SELECT number, hash FROM blocks WHERE number > $1
    ORDER BY number ASC',
    [safeBlock]
  )
```

```
  for (const row of recentBlocks.rows) {
    const chainBlock = await
    this.provider.getBlock(row.number)
```

```
    if (chainBlock.hash !== row.hash) {
      await this.handleReorg(row.number)
      return row.number
    }
  }
```

```
  return null
}
```

```
async handleReorg(fromBlock) {
  await this.database.transaction(async (client) => {
    await client.query(
      'DELETE FROM token_transfers WHERE block_number > =
      $1',
      [fromBlock]
    )
```

```
    await client.query(
      'DELETE FROM transactions WHERE block_number > = $1',
```

```
[fromBlock]
)
```

```
await client.query(
'DELETE FROM blocks WHERE number >= $1',
[fromBlock]
)
```

```
await this.recalculateBalances(client, fromBlock)
})
}
```

```
async recalculateBalances(client, fromBlock) {
await client.query('TRUNCATE token_balances')
```

```
await client.query(
`INSERT INTO token_balances (address, contract_address,
balance, last_updated_block)
SELECT
to_address,
contract_address,
SUM(value),
MAX(block_number)
FROM token_transfers
GROUP BY to_address, contract_address`
)
```

```
await client.query(
`UPDATE token_balances tb
SET balance = tb.balance - sub.sent
FROM (
SELECT from_address, contract_address, SUM(value) as sent
FROM token_transfers
WHERE from_address !=
'0x0000000000000000000000000000000000000000'
GROUP BY from_address, contract_address
) sub
```

```
WHERE tb.address = sub.from_address
AND tb.contract_address = sub.contract_address`
)
}
}
```

SECTION 5: DATABASE DESIGN PATTERNS

Choosing the Right Database

Different databases suit different blockchain data needs.

PostgreSQL excels for transactional data requiring ACID guarantees, complex queries with joins, and relational integrity. Use it for transaction histories, balance tracking, and user data.

MongoDB works well for event logs with varying schemas, document-oriented data like NFT metadata, and when you need flexible querying without strict schemas.

Redis serves as a cache layer for frequently accessed data like current prices, session storage, and real-time data like pending transactions.

TimescaleDB (PostgreSQL extension) handles time-series data efficiently. Use it for price history, trading volume over time, and metric tracking.

ClickHouse provides analytical queries at scale. Use it for aggregations across billions of rows, analytics dashboards, and historical analysis.

Partitioning Strategies

Blockchain data grows continuously. Partitioning keeps

queries fast.

```
CREATE TABLE token_transfers_partitioned (  
  id BIGSERIAL,  
  contract_address VARCHAR(42) NOT NULL,  
  block_number BIGINT NOT NULL,  
  transaction_hash VARCHAR(66) NOT NULL,  
  log_index INT NOT NULL,  
  from_address VARCHAR(42) NOT NULL,  
  to_address VARCHAR(42) NOT NULL,  
  value NUMERIC(78, 0) NOT NULL,  
  indexed_at TIMESTAMP DEFAULT NOW(),  
  PRIMARY KEY (id, block_number)  
) PARTITION BY RANGE (block_number);
```

```
CREATE TABLE token_transfers_p0 PARTITION OF  
token_transfers_partitioned  
FOR VALUES FROM (0) TO (1000000);
```

```
CREATE TABLE token_transfers_p1 PARTITION OF  
token_transfers_partitioned  
FOR VALUES FROM (1000000) TO (2000000);
```

```
CREATE TABLE token_transfers_p2 PARTITION OF  
token_transfers_partitioned  
FOR VALUES FROM (2000000) TO (3000000);
```

Automate partition creation as new blocks are indexed.

```
async function ensurePartition(database, blockNumber) {  
  const partitionSize = 1000000  
  const partitionNumber = Math.floor(blockNumber /  
partitionSize)  
  const startBlock = partitionNumber * partitionSize  
  const endBlock = startBlock + partitionSize  
  
  const partitionName = `token_transfers_p${partitionNumber}`
```

```

const exists = await database.query(
`SELECT 1 FROM pg_tables WHERE tablename = $1`,
[partitionName]
)

if (exists.rows.length === 0) {
  await database.query(
`CREATE TABLE ${partitionName} PARTITION OF
token_transfers_partitioned
FOR VALUES FROM (${startBlock}) TO (${endBlock})`
)
}
}

```

Caching Layer Design

Cache frequently accessed data to reduce database load.

```

import Redis from 'ioredis'

class BlockchainCache {
  constructor(redisUrl) {
    this.redis = new Redis(redisUrl)
  }

  async getTokenBalance(address, contractAddress) {
    const key = `balance:${contractAddress}:${address}`
    const cached = await this.redis.get(key)

    if (cached !== null) {
      return cached
    }

    return null
  }
}

```

```
}
```

```
async setTokenBalance(address, contractAddress, balance,  
blockNumber) {  
  const key = `balance:${contractAddress}:${address}`  
  const data = JSON.stringify({ balance, blockNumber })  
  await this.redis.set(key, data, 'EX', 60)  
}
```

```
async invalidateBalance(address, contractAddress) {  
  const key = `balance:${contractAddress}:${address}`  
  await this.redis.del(key)  
}
```

```
async getTokenPrice(symbol) {  
  const key = `price:${symbol}`  
  const cached = await this.redis.get(key)
```

```
  if (cached !== null) {  
    return JSON.parse(cached)  
  }
```

```
  return null  
}
```

```
async setTokenPrice(symbol, priceData) {  
  const key = `price:${symbol}`  
  await this.redis.set(key, JSON.stringify(priceData), 'EX', 30)  
}
```

```
async cacheTransaction(txHash, data) {  
  const key = `tx:${txHash}`  
  await this.redis.set(key, JSON.stringify(data), 'EX', 3600)  
}
```

```
async getTransaction(txHash) {  
  const key = `tx:${txHash}`
```

```

const cached = await this.redis.get(key)
return cached ? JSON.parse(cached) : null
}
}

```

SECTION 6: API ARCHITECTURE

RESTful API Design

Design APIs that expose your indexed data effectively.

```

import express from 'express'
import { ethers } from 'ethers'

```

```

const app = express()

```

```

app.get('/api/v1/address/:address/transactions', async (req,
res) => {

```

```

  const { address } = req.params

```

```

  const { page = 1, limit = 20, sort = 'desc' } = req.query

```

```

  if (!ethers.isAddress(address)) {

```

```

    return res.status(400).json({ error: 'Invalid address' })
  }

```

```

  const offset = (page - 1) * limit

```

```

  const result = await database.query(
    `SELECT hash, block_number, from_address, to_address,
value, gas_used, status
FROM transactions
WHERE from_address = $1 OR to_address = $1
ORDER BY block_number ${sort === 'asc' ? 'ASC' : 'DESC'}
LIMIT $2 OFFSET $3`,
    [address.toLowerCase(), limit, offset]
  )

```



```
const countResult = await database.query(
`SELECT COUNT(*) FROM transactions
WHERE from_address = $1 OR to_address = $1`,
[address.toLowerCase()]
)
```

```
res.json({
  data: result.rows,
  pagination: {
    page: Number(page),
    limit: Number(limit),
    total: Number(countResult.rows[0].count),
    totalPages: Math.ceil(countResult.rows[0].count / limit)
  }
})
})
```

```
app.get('/api/v1/address/:address/token-balances', async (req,
res) => {
  const { address } = req.params
```

```
  if (!ethers.isAddress(address)) {
    return res.status(400).json({ error: 'Invalid address' })
  }
```

```
  const result = await database.query(
`SELECT tb.contract_address, tb.balance, t.symbol, t.decimals,
t.name
FROM token_balances tb
LEFT JOIN tokens t ON tb.contract_address = t.address
WHERE tb.address = $1 AND tb.balance > 0
ORDER BY tb.balance DESC`,
[address.toLowerCase()]
)
```

```
const balances = result.rows.map(row => ({
```

```
token: {  
  address: row.contract_address,  
  name: row.name,  
  symbol: row.symbol,  
  decimals: row.decimals  
},  
balance: row.balance,  
formatted: formatUnits(row.balance, row.decimals || 18)  
}))
```

```
res.json({ data: balances })  
})
```

```
app.get('/api/v1/token/:address/transfers', async (req, res)  
=> {  
  const { address } = req.params  
  const { page = 1, limit = 50, from, to } = req.query
```

```
  const offset = (page - 1) * limit
```

```
  let query = `SELECT * FROM token_transfers WHERE  
contract_address = $1`  
  const params = [address.toLowerCase()]
```

```
  if (from) {  
    query += ` AND from_address = $$${params.length + 1}`  
    params.push(from.toLowerCase())  
  }
```

```
  if (to) {  
    query += ` AND to_address = $$${params.length + 1}`  
    params.push(to.toLowerCase())  
  }
```

```
  query += ` ORDER BY block_number DESC LIMIT $  
${params.length + 1} OFFSET $$${params.length + 2}`  
  params.push(limit, offset)
```

```
const result = await database.query(query, params)

res.json({ data: result.rows })
})

app.listen(3000)
```

WebSocket API for Real-Time Data

Provide real-time updates through WebSocket connections.

```
import { WebSocketServer } from 'ws'

class RealtimeServer {
  constructor(server) {
    this.wss = new WebSocketServer({ server })
    this.subscriptions = new Map()

    this.wss.on('connection', (ws) => {
      const clientId = this.generateClientId()

      this.subscriptions.set(clientId, {
        ws: ws,
        addresses: new Set(),
        tokens: new Set()
      })

      ws.on('message', (message) => {
        this.handleMessage(clientId, message)
      })

      ws.on('close', () => {
        this.subscriptions.delete(clientId)
      })
    })
  }
}
```

```
generateClientId() {  
  return Math.random().toString(36).substring(2, 15)  
}
```

```
handleMessage(clientId, message) {  
  const data = JSON.parse(message)  
  const subscription = this.subscriptions.get(clientId)
```

```
  switch (data.action) {  
    case 'subscribe_address':  
      subscription.addresses.add(data.address.toLowerCase())  
      break
```

```
    case 'unsubscribe_address':  
      subscription.addresses.delete(data.address.toLowerCase())  
      break
```

```
    case 'subscribe_token':  
      subscription.tokens.add(data.token.toLowerCase())  
      break
```

```
    case 'unsubscribe_token':  
      subscription.tokens.delete(data.token.toLowerCase())  
      break  
  }  
}
```

```
broadcastTransfer(transfer) {  
  const fromLower = transfer.from_address.toLowerCase()  
  const toLower = transfer.to_address.toLowerCase()  
  const contractLower =  
transfer.contract_address.toLowerCase()
```

```
  for (const [clientId, subscription] of this.subscriptions) {  
    const matchesAddress =  
subscription.addresses.has(fromLower) ||
```

```

subscription.addresses.has(toLower)

const matchesToken =
subscription.tokens.has(contractLower)

if (matchesAddress || matchesToken) {
subscription.ws.send(JSON.stringify({
type: 'transfer',
data: transfer
})))
}
}
}

broadcastNewBlock(block) {
const message = JSON.stringify({
type: 'block',
data: {
number: block.number,
hash: block.hash,
timestamp: block.timestamp,
transactionCount: block.transaction_count
}
})

for (const [clientId, subscription] of this.subscriptions) {
subscription.ws.send(message)
}
}
}

```

Rate Limiting and Authentication

Protect your API with proper rate limiting and authentication.

```
import rateLimit from 'express-rate-limit'
```

```
import { createHmac } from 'crypto'

const publicLimiter = rateLimit({
  windowMs: 60 * 1000,
  max: 30,
  message: { error: 'Too many requests' }
})

const authenticatedLimiter = rateLimit({
  windowMs: 60 * 1000,
  max: 300,
  keyGenerator: (req) => req.apiKey,
  message: { error: 'Rate limit exceeded' }
})

async function apiKeyAuth(req, res, next) {
  const apiKey = req.headers['x-api-key']

  if (!apiKey) {
    return next()
  }

  const keyRecord = await database.query(
    'SELECT user_id, tier FROM api_keys WHERE key_hash = $1'
    AND active = true',
    [hashApiKey(apiKey)]
  )

  if (keyRecord.rows.length > 0) {
    req.apiKey = apiKey
    req.userId = keyRecord.rows[0].user_id
    req.tier = keyRecord.rows[0].tier
  }

  next()
}
```

```
function hashApiKey(key) {
  return createHmac('sha256', process.env.API_KEY_SECRET)
    .update(key)
    .digest('hex')
}
```

```
app.use('/api/v1', apiKeyAuth)
```

```
app.use('/api/v1', (req, res, next) => {
  if (req.apiKey) {
    return authenticatedLimiter(req, res, next)
  }
  return publicLimiter(req, res, next)
})
```

SECTION 7: TRANSACTION MANAGEMENT

Nonce Management

Each account has a nonce that must increment with each transaction. Managing nonces correctly is critical for backend services sending multiple transactions.

```
class NonceManager {
  constructor(provider, address) {
    this.provider = provider
    this.address = address
    this.localNonce = null
    this.pendingTransactions = new Map()
    this.lock = false
  }

  async initialize() {
    this.localNonce = await this.provider.getTransactionCount(
      this.address,
      'pending'
    )
  }
}
```

```
)  
}
```

```
async getNextNonce() {  
  while (this.lock) {  
    await new Promise(resolve => setTimeout(resolve, 10))  
  }
```

```
  this.lock = true
```

```
  try {  
    const chainNonce = await  
    this.provider.getTransactionCount(  
      this.address,  
      'pending'  
    )
```

```
    if (chainNonce > this.localNonce) {  
      this.localNonce = chainNonce  
    }
```

```
    const nonce = this.localNonce  
    this.localNonce++
```

```
    return nonce  
  } finally {  
    this.lock = false  
  }  
}
```

```
async trackTransaction(nonce, txHash) {  
  this.pendingTransactions.set(nonce, txHash)  
}
```

```
async confirmTransaction(nonce) {  
  this.pendingTransactions.delete(nonce)  
}
```



```

async resetNonce() {
  this.localNonce = await this.provider.getTransactionCount(
    this.address,
    'latest'
  )
  this.pendingTransactions.clear()
}
}

```

Transaction Queue

Queue transactions to handle rate limits and ensure ordering.

```

class TransactionQueue {
  constructor(config) {
    this.provider = config.provider
    this.wallet = config.wallet
    this.nonceManager = new NonceManager(
      config.provider,
      config.wallet.address
    )
    this.queue = []
    this.processing = false
    this.onTransactionSent = config.onTransactionSent || (() => {})
    this.onTransactionConfirmed =
      config.onTransactionConfirmed || (() => {})
    this.onTransactionFailed = config.onTransactionFailed || (() => {})
  }

  async initialize() {
    await this.nonceManager.initialize()
  }
}

```

```
addTransaction(transaction, metadata = {}) {  
  return new Promise((resolve, reject) => {  
    this.queue.push({  
      transaction,  
      metadata,  
      resolve,  
      reject  
    })  
  })  
}
```

```
this.processQueue()  
})  
}
```

```
async processQueue() {  
  if (this.processing || this.queue.length === 0) {  
    return  
  }  
}
```

```
this.processing = true
```

```
while (this.queue.length > 0) {  
  const item = this.queue.shift()
```

```
  try {  
    const result = await this.sendTransaction(item)  
    item.resolve(result)  
  } catch (error) {  
    item.reject(error)  
  }  
}
```

```
await new Promise(resolve => setTimeout(resolve, 100))  
}
```

```
this.processing = false  
}
```

```
async sendTransaction(item) {
```

```

const { transaction, metadata } = item

const nonce = await this.nonceManager.getNextNonce()

const tx = {
  ...transaction,
  nonce: nonce
}

if (!tx.gasLimit) {
  tx.gasLimit = await this.provider.estimateGas(tx)
}

const feeData = await this.provider.getFeeData()
tx.maxFeePerGas = feeData.maxFeePerGas * BigInt(120) /
BigInt(100)
tx.maxPriorityFeePerGas = feeData.maxPriorityFeePerGas

const sentTx = await this.wallet.sendTransaction(tx)

this.nonceManager.trackTransaction(nonce, sentTx.hash)
this.onTransactionSent(sentTx.hash, metadata)

const receipt = await sentTx.wait()

this.nonceManager.confirmTransaction(nonce)

if (receipt.status === 1) {
  this.onTransactionConfirmed(receipt, metadata)
  return receipt
} else {
  const error = new Error('Transaction reverted')
  this.onTransactionFailed(error, receipt, metadata)
  throw error
}
}
}
}

```

Retry Logic with Exponential Backoff

Network issues and temporary failures require robust retry logic.

```
async function sendWithRetry(fn, maxRetries = 5) {
  let lastError

  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      return await fn()
    } catch (error) {
      lastError = error

      const isRetryable =
        error.code === 'NETWORK_ERROR' ||
        error.code === 'TIMEOUT' ||
        error.code === 'SERVER_ERROR' ||
        error.message.includes('nonce too low')

      if (!isRetryable) {
        throw error
      }

      const delay = Math.min(1000 * Math.pow(2, attempt),
        30000)
      const jitter = Math.random() * 1000

      await new Promise(resolve => setTimeout(resolve, delay +
        jitter))
    }
  }

  throw lastError
}
```

```

async function sendTransactionReliably(wallet, transaction) {
  return sendWithRetry(async () => {
    const tx = await wallet.sendTransaction(transaction)
    return tx.wait()
  })
}

```

SECTION 8: MONITORING AND OBSERVABILITY

Health Checks

Implement comprehensive health checks for blockchain services.

```

class HealthChecker {
  constructor(config) {
    this.provider = config.provider
    this.database = config.database
    this.redis = config.redis
  }

```

```

  async checkAll() {
    const results = await Promise.allSettled([
      this.checkProvider(),
      this.checkDatabase(),
      this.checkRedis(),
      this.checkIndexerLag()
    ])

```

```

    return {
      provider: this.formatResult(results[0]),
      database: this.formatResult(results[1]),
      redis: this.formatResult(results[2]),
      indexer: this.formatResult(results[3]),
      healthy: results.every(r => r.status === 'fulfilled' &&

```

```
r.value.healthy)
}
}
```

```
formatResult(result) {
  if (result.status === 'fulfilled') {
    return result.value
  }
  return { healthy: false, error: result.reason.message }
}
```

```
async checkProvider() {
  const startTime = Date.now()
  const blockNumber = await this.provider.getBlockNumber()
  const latency = Date.now() - startTime
```

```
  return {
    healthy: true,
    blockNumber,
    latency
  }
}
```

```
async checkDatabase() {
  const startTime = Date.now()
  await this.database.query('SELECT 1')
  const latency = Date.now() - startTime
```

```
  return {
    healthy: true,
    latency
  }
}
```

```
async checkRedis() {
  const startTime = Date.now()
  await this.redis.ping()
```

```
const latency = Date.now() - startTime
```

```
return {  
  healthy: true,  
  latency  
}  
}
```

```
async checkIndexerLag() {  
  const chainBlock = await this.provider.getBlockNumber()
```

```
  const result = await this.database.query(  
    'SELECT MAX(number) as last_block FROM blocks'  
  )
```

```
  const indexedBlock = result.rows[0]?.last_block || 0  
  const lag = chainBlock - indexedBlock
```

```
  return {  
    healthy: lag < 100,  
    chainBlock,  
    indexedBlock,  
    lag  
  }  
}  
}
```

```
app.get('/health', async (req, res) => {  
  const health = await healthChecker.checkAll()
```

```
  const statusCode = health.healthy ? 200 : 503
```

```
  res.status(statusCode).json(health)  
})
```

Metrics Collection

Track key metrics for blockchain services.

```
import { Counter, Histogram, Gauge, Registry } from 'prom-client'
```

```
const registry = new Registry()
```

```
const rpcRequestsTotal = new Counter({  
  name: 'rpc_requests_total',  
  help: 'Total number of RPC requests',  
  labelNames: ['method', 'status'],  
  registers: [registry]  
})
```

```
const rpcRequestDuration = new Histogram({  
  name: 'rpc_request_duration_seconds',  
  help: 'RPC request duration in seconds',  
  labelNames: ['method'],  
  buckets: [0.01, 0.05, 0.1, 0.5, 1, 2, 5],  
  registers: [registry]  
})
```

```
const indexerBlockLag = new Gauge({  
  name: 'indexer_block_lag',  
  help: 'Number of blocks behind chain head',  
  registers: [registry]  
})
```

```
const transactionsSent = new Counter({  
  name: 'transactions_sent_total',  
  help: 'Total transactions sent',  
  labelNames: ['status'],  
  registers: [registry]  
})
```

```
class MetricsProvider {
```



```

constructor(baseProvider) {
  this.baseProvider = baseProvider
}

async request(method, params) {
  const end = rpcRequestDuration.startTimer({ method })

  try {
    const result = await this.baseProvider.request(method,
params)
    rpcRequestsTotal.inc({ method, status: 'success' })
    return result
  } catch (error) {
    rpcRequestsTotal.inc({ method, status: 'error' })
    throw error
  } finally {
    end()
  }
}
}

app.get('/metrics', async (req, res) => {
  res.set('Content-Type', registry.contentType)
  res.send(await registry.metrics())
})

```

Alerting Configuration

Set up alerts for critical conditions.

```

class AlertManager {
  constructor(config) {
    this.webhookUrl = config.webhookUrl
    this.thresholds = config.thresholds
  }
}

```

```
async checkAndAlert(metrics) {  
  const alerts = []
```

```
  if (metrics.indexerLag > this.thresholds.maxIndexerLag) {  
    alerts.push({  
      severity: 'critical',  
      message: `Indexer lag is ${metrics.indexerLag} blocks`,  
      metric: 'indexer_lag',  
      value: metrics.indexerLag  
    })  
  }  
}
```

```
  if (metrics.rpcLatency > this.thresholds.maxRpcLatency) {  
    alerts.push({  
      severity: 'warning',  
      message: `RPC latency is ${metrics.rpcLatency}ms`,  
      metric: 'rpc_latency',  
      value: metrics.rpcLatency  
    })  
  }  
}
```

```
  if (metrics.pendingTransactions >  
    this.thresholds.maxPendingTx) {  
    alerts.push({  
      severity: 'warning',  
      message: `${metrics.pendingTransactions} transactions  
pending`,  
      metric: 'pending_transactions',  
      value: metrics.pendingTransactions  
    })  
  }  
}
```

```
  if (metrics.walletBalance <  
    this.thresholds.minWalletBalance) {  
    alerts.push({  
      severity: 'critical',  
      message: `Hot wallet balance low: ${metrics.walletBalance}
```

```

    ETH`,
    metric: 'wallet_balance',
    value: metrics.walletBalance
  })
}

for (const alert of alerts) {
  await this.sendAlert(alert)
}

return alerts
}

async sendAlert(alert) {
  await fetch(this.webhookUrl, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      text: `[${alert.severity.toUpperCase()}] ${alert.message}`,
      severity: alert.severity,
      metric: alert.metric,
      value: alert.value,
      timestamp: new Date().toISOString()
    })
  })
}
}

```

SECTION 9: SECURITY BEST PRACTICES

Private Key Management

Never store private keys in code, environment variables visible to logs, or unencrypted databases.

```
import { KMSClient, DecryptCommand } from '@aws-sdk/
```

client-kms'

```
class SecureKeyManager {  
  constructor(config) {  
    this.kmsClient = new KMSSClient({ region: config.region })  
    this.keyId = config.keyId  
    this.cachedKey = null  
    this.cacheExpiry = null  
  }
```

```
  async getPrivateKey() {  
    if (this.cachedKey && this.cacheExpiry > Date.now()) {  
      return this.cachedKey  
    }
```

```
    const encryptedKey =  
    process.env.ENCRYPTED_PRIVATE_KEY
```

```
    const command = new DecryptCommand({  
      KeyId: this.keyId,  
      CiphertextBlob: Buffer.from(encryptedKey, 'base64')  
    })
```

```
    const response = await this.kmsClient.send(command)  
    const privateKey =  
    Buffer.from(response.Plaintext).toString('utf-8')
```

```
    this.cachedKey = privateKey  
    this.cacheExpiry = Date.now() + 5 * 60 * 1000
```

```
    return privateKey  
  }
```

```
  async createWallet(provider) {  
    const privateKey = await this.getPrivateKey()  
    return new ethers.Wallet(privateKey, provider)  
  }
```

```
}
```

For high-value operations use hardware security modules or multi-signature schemes.

Input Validation

Validate all inputs especially addresses and amounts.

```
import { ethers } from 'ethers'

function validateAddress(address) {
  if (!address || typeof address !== 'string') {
    throw new Error('Address is required')
  }

  if (!ethers.isAddress(address)) {
    throw new Error('Invalid Ethereum address')
  }

  return ethers.getAddress(address)
}

function validateAmount(amount, decimals = 18) {
  if (amount === undefined || amount === null) {
    throw new Error('Amount is required')
  }

  const value = BigInt(amount)

  if (value < 0n) {
    throw new Error('Amount must be non-negative')
  }

  if (value > ethers.MaxUint256) {
    throw new Error('Amount exceeds maximum')
  }
}
```

```

    return value
  }

  function validateTransactionHash(hash) {
    if (!hash || typeof hash !== 'string') {
      throw new Error('Transaction hash is required')
    }

    if (!/^0x[a-fA-F0-9]{64}$/.test(hash)) {
      throw new Error('Invalid transaction hash format')
    }

    return hash.toLowerCase()
  }

```

Secure RPC Configuration

Protect your RPC endpoints and API keys.

```

class SecureProvider {
  constructor(config) {
    this.providers = config.providers.map(url =>
      new ethers.JsonRpcProvider(url)
    )
    this.currentIndex = 0
    this.maxRetries = config.maxRetries || 3
  }

  async send(method, params) {
    let lastError

    for (let i = 0; i < this.maxRetries; i++) {
      const provider = this.providers[this.currentIndex]

      try {

```

```

return await provider.send(method, params)
} catch (error) {
  lastError = error
  this.currentIndex = (this.currentIndex + 1) %
this.providers.length
}
}

throw lastError
}

async getBlockNumber() {
return this.send('eth_blockNumber', [])
}
}

const secureProvider = new SecureProvider({
  providers: [
    process.env.RPC_URL_PRIMARY,
    process.env.RPC_URL_SECONDARY,
    process.env.RPC_URL_FALLBACK
  ],
  maxRetries: 3
})

```

SECTION 10: PRODUCTION DEPLOYMENT

Docker Configuration

Containerize your blockchain services.

FROM node:20-alpine

WORKDIR /app

COPY package*.json ./

RUN npm ci --only=production

COPY . .

ENV NODE_ENV=production

USER node

EXPOSE 3000

HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \

CMD wget --no-verbose --tries=1 --spider http://localhost:3000/health || exit 1

CMD ["node", "src/index.js"]

Kubernetes Deployment

Deploy with proper resource limits and scaling.

apiVersion: apps/v1

kind: Deployment

metadata:

name: blockchain-indexer

spec:

replicas: 2

selector:

matchLabels:

app: blockchain-indexer

template:

metadata:

labels:

app: blockchain-indexer

spec:

containers:
- name: indexer
image: blockchain-indexer:latest
ports:
- containerPort: 3000
resources:
requests:
memory: "512Mi"
cpu: "250m"
limits:
memory: "1Gi"
cpu: "500m"
env:
- name: DATABASE_URL
valueFrom:
secretKeyRef:
name: blockchain-secrets
key: database-url
- name: RPC_URL
valueFrom:
secretKeyRef:
name: blockchain-secrets
key: rpc-url
livenessProbe:
httpGet:
path: /health
port: 3000
initialDelaySeconds: 30
periodSeconds: 10
readinessProbe:
httpGet:
path: /health
port: 3000
initialDelaySeconds: 5
periodSeconds: 5

Environment Configuration

Manage configuration across environments.

```
const config = {
  development: {
    rpcUrl: 'http://localhost:8545',
    chainId: 31337,
    database: 'postgresql://localhost/blockchain_dev',
    redis: 'redis://localhost:6379',
    logLevel: 'debug'
  },
  staging: {
    rpcUrl: process.env.RPC_URL,
    chainId: 11155111,
    database: process.env.DATABASE_URL,
    redis: process.env.REDIS_URL,
    logLevel: 'info'
  },
  production: {
    rpcUrl: process.env.RPC_URL,
    chainId: 1,
    database: process.env.DATABASE_URL,
    redis: process.env.REDIS_URL,
    logLevel: 'warn'
  }
}

function getConfig() {
  const env = process.env.NODE_ENV || 'development'
  return config[env]
}
```

CHAPTER SUMMARY

This chapter covered the complete backend infrastructure for

blockchain applications.

We explored Web3 libraries including ethers.js for JavaScript and web3.py for Python. These libraries abstract JSON-RPC complexity and provide type-safe contract interaction.

RPC methods are the foundation of blockchain communication. Understanding `eth_call` versus `eth_sendRawTransaction`, batch requests, and Multicall contracts enables efficient data access.

Event listening through WebSocket connections provides real-time updates. Production listeners need reconnection logic, checkpoint management, and catch-up mechanisms for downtime.

Indexing transforms slow blockchain queries into fast database queries. Proper schema design, partitioning, and reorg handling are essential for reliable indexers.

Database design patterns including PostgreSQL for transactions, Redis for caching, and proper partitioning strategies keep queries fast as data grows.

API architecture exposes indexed data through REST and WebSocket interfaces with proper rate limiting and authentication.

Transaction management requires careful nonce handling, queuing, and retry logic for reliable transaction submission.

Monitoring and observability through health checks, metrics, and alerting ensure service reliability.

Security practices including key management, input validation, and secure RPC configuration protect against attacks.

With backend services in place your application can process blockchain data at scale. The final chapter covers testing and deployment workflows to bring everything to production.

CHAPTER 10: TESTING AND DEPLOYMENT

Writing smart contracts is only the beginning. The real challenge is ensuring your code works correctly before deploying immutable logic that handles real value. This chapter covers the complete testing and deployment lifecycle from local development through mainnet launch and ongoing monitoring.

SECTION 1: LOCAL DEVELOPMENT ENVIRONMENTS

Why Local Development Matters

Deploying to testnets costs time. Each transaction requires waiting for block confirmations. Faucet tokens are limited. Network congestion affects test reliability.

Local development environments give you instant transactions, unlimited test ETH, full control over blockchain state, and the ability to fork mainnet data. You can reset state, manipulate time, and test scenarios impossible on live networks.

Hardhat Setup

Hardhat is the most widely used Ethereum development environment. It provides a local node, testing framework, deployment scripts, and plugin ecosystem.

Initialize a new Hardhat project.

```
npm init -y
```

```
npm install --save-dev hardhat @nomicfoundation/hardhat-toolbox
```

```
npx hardhat init
```

Select "Create a JavaScript project" when prompted. This generates a project structure with sample contracts, tests, and configuration.

The `hardhat.config.js` file configures networks, compiler settings, and plugins.

```
require("@nomicfoundation/hardhat-toolbox")
```

```
module.exports = {  
  solidity: {  
    version: "0.8.24",  
    settings: {  
      optimizer: {  
        enabled: true,  
        runs: 200  
      },  
      viaIR: true  
    },  
  },  
  networks: {  
    hardhat: {  
      chainId: 31337,  
      mining: {  
        auto: true,  
        interval: 0  
      },  
    },  
    localhost: {  
      url: "http://127.0.0.1:8545",  
      chainId: 31337  
    },  
  },  
}
```

```

sepolia: {
  url: process.env.SEPOLIA_RPC_URL,
  accounts: process.env.PRIVATE_KEY ?
[process.env.PRIVATE_KEY] : [],
  chainId: 11155111
},
mainnet: {
  url: process.env.MAINNET_RPC_URL,
  accounts: process.env.PRIVATE_KEY ?
[process.env.PRIVATE_KEY] : [],
  chainId: 1
}
},
etherscan: {
  apiKey: process.env.ETHERSCAN_API_KEY
},
gasReporter: {
  enabled: true,
  currency: "USD"
}
}

```

Running Hardhat Node

Start a local blockchain with pre-funded accounts.

`npx hardhat node`

Breaking down:

`npx hardhat node` - Starts local Ethereum node

`npx` - Node package executor. Runs locally installed packages

`hardhat` - Development framework

`node` - Subcommand to start local blockchain

This starts a JSON-RPC server at `http://127.0.0.1:8545` with

20 accounts each holding 10000 ETH. The terminal displays account addresses and private keys.

Connect MetaMask to localhost:8545 and import one of the displayed private keys to interact with your local blockchain.

Foundry Setup

Foundry is a blazingly fast toolkit written in Rust. It compiles faster, runs tests faster, and provides powerful debugging tools.

Install Foundry using the installer script.

```
curl -L https://foundry.paradigm.xyz | bash  
foundryup
```

Breaking down:

```
curl -L https://foundry.paradigm.xyz - Downloads installer  
script  
| bash - Pipes output to bash for execution  
foundryup - Updates Foundry to latest version
```

Initialize a new Foundry project.

```
forge init my-project  
cd my-project
```

Breaking down:

```
forge init my-project - Creates new Foundry project  
forge - Foundry build tool  
init - Initialization subcommand  
my-project - Directory name
```

Foundry uses a different project structure. Source contracts go

in src/, tests in test/, and deployment scripts in script/.

The foundry.toml file configures the project.

```
[profile.default]
src = "src"
out = "out"
libs = ["lib"]
solc = "0.8.24"
optimizer = true
optimizer_runs = 200
via_ir = true

[rpc_endpoints]
sepolia = "${SEPOLIA_RPC_URL}"
mainnet = "${MAINNET_RPC_URL}"

[etherscan]
sepolia = { key = "${ETHERSCAN_API_KEY}" }
mainnet = { key = "${ETHERSCAN_API_KEY}" }
```

Running Anvil

Anvil is Foundry's local node equivalent to Hardhat's network.

anvil

Breaking down:

anvil - Local Ethereum node from Foundry toolkit

Anvil starts with similar features to Hardhat node including pre-funded accounts and instant mining. It supports advanced features like forking, impersonation, and time manipulation.

Forking Mainnet

Both Hardhat and Foundry can fork mainnet state, creating a local copy of all contracts and balances at a specific block.

With Hardhat add forking configuration.

```
networks: {  
  hardhat: {  
    forking: {  
      url: process.env.MAINNET_RPC_URL,  
      blockNumber: 19000000  
    }  
  }  
}
```

With Anvil use the fork flag.

```
anvil --fork-url $MAINNET_RPC_URL --fork-block-number  
19000000
```

Breaking down:

anvil - Local node
--fork-url \$MAINNET_RPC_URL - RPC endpoint to fork from
--fork-block-number 19000000 - Specific block number for
reproducibility

Forking lets you test interactions with deployed protocols. You can trade on Uniswap, borrow from Aave, or test against any mainnet contract using their actual deployed state.

SECTION 2: WRITING SMART CONTRACT TESTS

Hardhat Test Structure

Hardhat tests use Mocha and Chai with additional matchers for blockchain assertions.

```
const { expect } = require("chai")
const { ethers } = require("hardhat")
const { loadFixture } = require("@nomicfoundation/hardhat-toolbox/network-helpers")
```

```
describe("Token", function () {
  async function deployTokenFixture() {
    const [owner, addr1, addr2] = await ethers.getSigners()
```

```
    const Token = await ethers.getContractFactory("Token")
    const token = await Token.deploy("MyToken", "MTK",
    1000000)
```

```
    return { token, owner, addr1, addr2 }
  }
```

```
  describe("Deployment", function () {
    it("should set the correct name and symbol", async function () {
      const { token } = await loadFixture(deployTokenFixture)
```

```
      expect(await token.name()).to.equal("MyToken")
      expect(await token.symbol()).to.equal("MTK")
    })
  })
```

```
  it("should assign total supply to owner", async function () {
    const { token, owner } = await
    loadFixture(deployTokenFixture)
```

```
    const ownerBalance = await token.balanceOf(owner.address)
    expect(await token.totalSupply()).to.equal(ownerBalance)
  })
})
```

```
describe("Transfers", function () {
  it("should transfer tokens between accounts", async function
```

```

() {
  const { token, owner, addr1, addr2 } = await
loadFixture(deployTokenFixture)

  await token.transfer(addr1.address, 100)
  expect(await token.balanceOf(addr1.address)).to.equal(100)

  await token.connect(addr1).transfer(addr2.address, 50)
  expect(await token.balanceOf(addr2.address)).to.equal(50)
  expect(await token.balanceOf(addr1.address)).to.equal(50)
}

it("should fail if sender lacks balance", async function () {
  const { token, owner, addr1 } = await
loadFixture(deployTokenFixture)

  await expect(
    token.connect(addr1).transfer(owner.address, 1)
  ).to.be.revertedWithCustomError(token, "InsufficientBalance")
})

it("should emit Transfer event", async function () {
  const { token, owner, addr1 } = await
loadFixture(deployTokenFixture)

  await expect(token.transfer(addr1.address, 100))
    .to.emit(token, "Transfer")
    .withArgs(owner.address, addr1.address, 100)
  })
  })
  })

```

The `loadFixture` function optimizes tests by taking a snapshot after the first run and reverting to it for subsequent tests rather than redeploying.

Foundry Test Structure

Foundry tests are written in Solidity using the forge-std library.

```
pragma solidity ^0.8.24;

import {Test, console} from "forge-std/Test.sol";
import {Token} from "../src/Token.sol";

contract TokenTest is Test {
    Token public token;
    address public owner;
    address public addr1;
    address public addr2;

    function setUp() public {
        owner = address(this);
        addr1 = makeAddr("addr1");
        addr2 = makeAddr("addr2");

        token = new Token("MyToken", "MTK", 1000000);
    }

    function test_NameAndSymbol() public view {
        assertEq(token.name(), "MyToken");
        assertEq(token.symbol(), "MTK");
    }

    function test_OwnerBalance() public view {
        assertEq(token.balanceOf(owner), token.totalSupply());
    }

    function test_Transfer() public {
        token.transfer(addr1, 100);
        assertEq(token.balanceOf(addr1), 100);
    }
}
```

```
vm.prank(addr1);  
token.transfer(addr2, 50);
```

```
assertEq(token.balanceOf(addr1), 50);  
assertEq(token.balanceOf(addr2), 50);  
}
```

```
function test_TransferInsufficientBalance() public {  
    vm.prank(addr1);  
    vm.expectRevert(Token.InsufficientBalance.selector);  
    token.transfer(owner, 1);  
}
```

```
function test_TransferEmitsEvent() public {  
    vm.expectEmit(true, true, false, true);  
    emit Token.Transfer(owner, addr1, 100);  
    token.transfer(addr1, 100);  
}
```

```
function testFuzz_Transfer(uint256 amount) public {  
    amount = bound(amount, 0, token.balanceOf(owner));
```

```
    uint256 ownerBefore = token.balanceOf(owner);
```

```
    token.transfer(addr1, amount);
```

```
    assertEq(token.balanceOf(owner), ownerBefore - amount);  
    assertEq(token.balanceOf(addr1), amount);  
}  
}
```

The vm object provides cheat codes for manipulating blockchain state. `vm.prank` sets `msg.sender` for the next call. `vm.expectRevert` expects the next call to revert.

Testing Time-Dependent Logic

Many contracts depend on timestamps. Test frameworks let you manipulate time.

In Hardhat use time helpers.

```
const { time } = require("@nomicfoundation/hardhat-toolbox/network-helpers")
```

```
it("should allow unlock after time passes", async function () {  
  const { lock, unlockTime } = await  
  loadFixture(deployLockFixture)
```

```
  await time.increaseTo(unlockTime)
```

```
  await expect(lock.withdraw()).not.to.be.reverted  
})
```

```
it("should prevent early withdrawal", async function () {  
  const { lock } = await loadFixture(deployLockFixture)
```

```
  await expect(lock.withdraw()).to.be.revertedWith("Too  
early")  
})
```

In Foundry use `vm.warp`.

```
function test_UnlockAfterTime() public {  
  vm.warp(lock.unlockTime());
```

```
  lock.withdraw();  
}
```

```
function test_PreventEarlyWithdrawal() public {  
  vm.expectRevert("Too early");  
  lock.withdraw();  
}
```

Testing with Forked State

Test against real protocol deployments by forking mainnet.

```
const { expect } = require("chai")
const { ethers } = require("hardhat")

describe("Uniswap Integration", function () {
  const UNISWAP_ROUTER =
    "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D"
  const WETH =
    "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2"
  const USDC =
    "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48"

  it("should swap ETH for USDC", async function () {
    const [signer] = await ethers.getSigners()

    const router = await ethers.getContractAt(
      "IUniswapV2Router02",
      UNISWAP_ROUTER
    )

    const usdc = await ethers.getContractAt("IERC20", USDC)
    const balanceBefore = await usdc.balanceOf(signer.address)

    const amountIn = ethers.parseEther("1")
    const deadline = Math.floor(Date.now() / 1000) + 3600

    await router.swapExactETHForTokens(
      0,
      [WETH, USDC],
      signer.address,
      deadline,
      { value: amountIn }
    )
```

```

const balanceAfter = await usdc.balanceOf(signer.address)
expect(balanceAfter).to.be.gt(balanceBefore)
})
})

```

Fuzz Testing

Fuzz testing generates random inputs to find edge cases your manual tests miss.

Foundry has built-in fuzzing.

```

function testFuzz_Approve(address spender, uint256 amount)
public {
    vm.assume(spender != address(0));

    token.approve(spender, amount);

    assertEq(token.allowance(address(this), spender), amount);
}

```

```

function testFuzz_TransferFrom(
    address from,
    address to,
    uint256 approveAmount,
    uint256 transferAmount
) public {
    vm.assume(from != address(0));
    vm.assume(to != address(0));
    vm.assume(from != to);
    transferAmount = bound(transferAmount, 0,
        approveAmount);

    deal(address(token), from, approveAmount);
}

```



```

vm.prank(from);
token.approve(address(this), approveAmount);

token.transferFrom(from, to, transferAmount);

assertEq(token.balanceOf(to), transferAmount);
assertEq(token.allowance(from, address(this)),
approveAmount - transferAmount);
}

```

The `vm.assume` function skips test runs with invalid inputs. The `bound` function constrains values to valid ranges. Foundry runs many iterations with random inputs.

Invariant Testing

Invariant tests verify properties that should always hold regardless of what operations occur.

```

contract TokenInvariantTest is Test {
    Token public token;
    Handler public handler;

    function setUp() public {
        token = new Token("Test", "TST", 1000000);
        handler = new Handler(token);

        targetContract(address(handler));
    }

    function invariant_totalSupplyConstant() public view {
        assertEq(token.totalSupply(), 1000000);
    }

    function invariant_balancesSumToTotal() public view {
        uint256 sum = token.balanceOf(address(handler));

```

```

sum += token.balanceOf(handler.actor1());
sum += token.balanceOf(handler.actor2());
assertLe(sum, token.totalSupply());
}
}

```

contract Handler is Test {

```

    Token public token;
    address public actor1;
    address public actor2;

```

```

    constructor(Token _token) {
        token = _token;
        actor1 = makeAddr("actor1");
        actor2 = makeAddr("actor2");
    }

```

```

    token.transfer(address(this), token.totalSupply() / 2);
}

```

```

    function transfer(uint256 amount, bool toActor1) public {
        address to = toActor1 ? actor1 : actor2;
        amount = bound(amount, 0, token.balanceOf(address(this)));
        token.transfer(to, amount);
    }

```

function transferBetweenActors(uint256 amount, bool fromActor1) public {

```

    address from = fromActor1 ? actor1 : actor2;
    address to = fromActor1 ? actor2 : actor1;
    amount = bound(amount, 0, token.balanceOf(from));

```

```

    vm.prank(from);
    token.transfer(to, amount);
}
}

```

SECTION 3: CODE COVERAGE AND ANALYSIS

Coverage Reports

Code coverage identifies untested code paths.

In Hardhat install the coverage plugin if not included.

```
npm install hardhat-coverage
```

This generates a coverage report showing which lines, branches, and functions your tests execute.

In Foundry use the coverage command.

```
forge coverage
```

Breaking down:

forge coverage - Runs tests and generates coverage report

For detailed HTML reports.

```
forge coverage --report lcov
```

Breaking down:

forge coverage --report lcov - Generates LCOV format report

--report lcov - Output format flag. lcov = LCOV format

Then use genhtml to create browsable HTML.

```
genhtml -o coverage lcov.info
```

Breaking down:

genhtml -o coverage lcov.info - Generates HTML from LCOV

genhtml - HTML generator from LCOV

-o coverage - Output directory
lcov.info - Input LCOV file

Static Analysis

Static analysis tools find vulnerabilities without running code.

Slither is the most comprehensive static analyzer for Solidity.

```
pip install slither-analyzer  
slither .
```

Breaking down:

```
pip install slither-analyzer - Installs Slither  
slither . - Analyzes current directory
```

Slither detects reentrancy, uninitialized storage, dangerous delegatecalls, and many other vulnerabilities. Configure detection sensitivity in slither.config.json.

```
{  
  "filter_paths": "node_modules|lib",  
  "exclude_informational": false,  
  "exclude_low": false,  
  "exclude_medium": false,  
  "exclude_high": false  
}
```

Gas Optimization Analysis

Foundry provides gas snapshots for comparison.

```
forge snapshot
```

Breaking down:

forge snapshot - Creates gas snapshot of all tests

This creates .gas-snapshot file. After changes compare with.

forge snapshot --diff

Breaking down:

forge snapshot --diff - Compares current gas usage to snapshot

The report shows which functions use more or less gas after your changes.

SECTION 4: TESTNET DEPLOYMENT

Choosing a Testnet

Different testnets serve different purposes.

Sepolia is the primary Ethereum testnet. It uses proof-of-stake and closely mirrors mainnet behavior. Use Sepolia for most testing.

Holesky is designed for staking and infrastructure testing with more validators than Sepolia.

Goerli is deprecated and should not be used for new projects.

For Layer 2 testing each network has its own testnet. Arbitrum Sepolia, Optimism Sepolia, and Base Sepolia mirror their mainnet counterparts.

Getting Testnet ETH

Faucets distribute free testnet ETH. Popular options include.

Alchemy faucet at faucet.alchemy.com requires an Alchemy account.

Infura faucet at infura.io/faucet requires an Infura account.

QuickNode faucet at faucet.quicknode.com.

Most faucets limit claims to once per day per address. Request ETH before you need it.

Deployment Scripts with Hardhat

Create deployment scripts in the scripts directory.

```
const { ethers, run, network } = require("hardhat")

async function main() {
  console.log("Deploying to network:", network.name)

  const [deployer] = await ethers.getSigners()
  console.log("Deploying with account:", deployer.address)

  const balance = await
ethers.provider.getBalance(deployer.address)
  console.log("Account balance:", ethers.formatEther(balance),
"ETH")

  const Token = await ethers.getContractFactory("Token")
  const token = await Token.deploy(
    "MyToken",
    "MTK",
    ethers.parseEther("1000000")
  )

  await token.waitForDeployment()
  const tokenAddress = await token.getAddress()

  console.log("Token deployed to:", tokenAddress)
```

```

if (network.name !== "hardhat" && network.name !==
"localhost") {
  console.log("Waiting for block confirmations...")
  await token.deployedTransaction().wait(5)

  console.log("Verifying contract...")
  try {
    await run("verify:verify", {
      address: tokenAddress,
      constructorArguments: [
        "MyToken",
        "MTK",
        ethers.parseEther("1000000")
      ]
    })
    console.log("Contract verified!")
  } catch (error) {
    console.log("Verification failed:", error.message)
  }
}

return { token, tokenAddress }
}

```

```

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error)
    process.exit(1)
  })

```

Run deployment with environment variables.

```

SEPOLIA_RPC_URL=https://... PRIVATE_KEY=0x... npx
hardhat run scripts/deploy.js --network sepolia

```

Breaking down:

SEPOLIA_RPC_URL=https://... - Sets RPC URL environment variable

PRIVATE_KEY=0x... - Sets deployer private key

npx hardhat run scripts/deploy.js --network sepolia - Runs deployment script

--network sepolia - Target network flag

Deployment Scripts with Foundry

Foundry uses Solidity scripts in the script directory.

```
pragma solidity ^0.8.24;
```

```
import {Script, console} from "forge-std/Script.sol";  
import {Token} from "../src/Token.sol";
```

```
contract DeployToken is Script {  
    function run() public returns (Token) {  
        uint256 deployerPrivateKey = vm.envUint("PRIVATE_KEY");
```

```
        vm.startBroadcast(deployerPrivateKey);
```

```
        Token token = new Token(  
            "MyToken",  
            "MTK",  
            1000000 ether  
        );
```

```
        console.log("Token deployed to:", address(token));
```

```
        vm.stopBroadcast();
```

```
        return token;  
    }  
}
```


Run deployment with forge script.

```
forge script script/DeployToken.s.sol:DeployToken --rpc-url  
$SEPOLIA_RPC_URL --broadcast --verify
```

Breaking down:

forge script script/DeployToken.s.sol:DeployToken - Runs
specified script
script/DeployToken.s.sol - Script file path
:DeployToken - Contract name within file
--rpc-url \$SEPOLIA_RPC_URL - RPC endpoint
--broadcast - Actually sends transactions (dry run without
this)
--verify - Verify contract on Etherscan

Managing Deployment Addresses

Track deployed addresses across networks.

```
{  
  "sepolia": {  
    "token": "0x1234567890abcdef1234567890abcdef12345678",  
    "nft": "0xabcdef1234567890abcdef1234567890abcdef12",  
    "deployedAt": "2026-01-04T10:30:00Z",  
    "deployer":  
    "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE1b"  
  },  
  "mainnet": {  
    "token": null,  
    "nft": null,  
    "deployedAt": null,  
    "deployer": null  
  }  
}
```

Automated scripts should update this file after successful deployments.

SECTION 5: MAINNET DEPLOYMENT

Pre-Deployment Checklist

Before deploying to mainnet verify every item.

Security: Have contracts been audited? Are all audit findings resolved? Has the code changed since the audit?

Testing: Do all tests pass? Is coverage above 90%? Have you tested on a forked mainnet?

Access Control: Are admin functions properly restricted? Are ownership transfers tested? Can the contract be paused if needed?

Gas: Is gas usage acceptable? Have you tested with realistic gas prices? Are there any unbounded loops?

Dependencies: Are external contract addresses correct? Have you verified dependency contracts? What happens if dependencies fail?

Funds: Is the deployer wallet funded? Have you estimated deployment cost? Is there buffer for failed transactions?

Documentation: Is deployment documented? Are upgrade procedures documented? Is emergency response documented?

Estimating Deployment Cost

Calculate deployment cost before executing.

```

async function estimateDeploymentCost() {
  const Token = await ethers.getContractFactory("Token")

  const deploymentData = Token.getDeployTransaction(
    "MyToken",
    "MTK",
    ethers.parseEther("1000000")
  )

  const estimatedGas = await
ethers.provider.estimateGas(deploymentData)
  const feeData = await ethers.provider.getFeeData()

  const maxCost = estimatedGas * feeData.maxFeePerGas
  const likelyCost = estimatedGas * (feeData.maxFeePerGas +
feeData.maxPriorityFeePerGas) / 2n

  console.log("Estimated gas:", estimatedGas.toString())
  console.log("Max cost:", ethers.formatEther(maxCost), "ETH")
  console.log("Likely cost:", ethers.formatEther(likelyCost),
"ETH")

  const ethPrice = 3000
  console.log("Likely cost USD:",
Number(ethers.formatEther(likelyCost)) * ethPrice)
}

```

Mainnet Deployment Script

Production deployment scripts need additional safety checks.

```

const { ethers, network } = require("hardhat")
const readline = require("readline")

async function confirmDeployment() {

```

```
const rl = readline.createInterface({  
  input: process.stdin,  
  output: process.stdout  
})
```

```
return new Promise((resolve) => {  
  rl.question("Type 'DEPLOY' to confirm mainnet deployment: ",  
(answer) => {  
    rl.close()  
    resolve(answer === "DEPLOY")  
  })  
})  
}
```

```
async function main() {  
  if (network.name !== "mainnet") {  
    throw new Error("This script is for mainnet only")  
  }
```

```
  console.log("=== MAINNET DEPLOYMENT ===")  
  console.log("Network:", network.name)  
  console.log("Chain ID:", network.config.chainId)
```

```
  const [deployer] = await ethers.getSigners()  
  console.log("Deployer:", deployer.address)
```

```
  const balance = await  
  ethers.provider.getBalance(deployer.address)  
  console.log("Balance:", ethers.formatEther(balance), "ETH")
```

```
  if (balance < ethers.parseEther("0.1")) {  
    throw new Error("Insufficient balance for deployment")  
  }
```

```
  const confirmed = await confirmDeployment()  
  if (!confirmed) {  
    console.log("Deployment cancelled")
```

```
process.exit(0)
}
```

```
console.log("\nDeploying contracts...")
```

```
const Token = await ethers.getContractFactory("Token")
const token = await Token.deploy(
  "MyToken",
  "MTK",
  ethers.parseEther("1000000"),
  {
    maxFeePerGas: ethers.parseGwei("50"),
    maxPriorityFeePerGas: ethers.parseGwei("2")
  }
)
```

```
console.log("Transaction hash:",
token.deploymentTransaction().hash)
console.log("Waiting for confirmation...")
```

```
await token.waitForDeployment()
```

```
const address = await token.getAddress()
console.log("\n=== DEPLOYMENT SUCCESSFUL ===")
console.log("Token address:", address)
console.log("Block:",
token.deploymentTransaction().blockNumber)
```

```
const fs = require("fs")
const deployments =
JSON.parse(fs.readFileSync("deployments.json"))
deployments.mainnet = {
  token: address,
  deployedAt: new Date().toISOString(),
  deployer: deployer.address,
  txHash: token.deploymentTransaction().hash
}
```

```
fs.writeFileSync("deployments.json",
JSON.stringify(deployments, null, 2))
}
```

```
main()
.then(() => process.exit(0))
.catch((error) => {
  console.error(error)
  process.exit(1)
})
```

SECTION 6: CONTRACT VERIFICATION

Why Verify Contracts

Verified contracts show their source code on block explorers. Users can read the code, understand what they are interacting with, and trust the contract more. Unverified contracts are opaque bytecode that raises suspicion.

Verification also enables the Read and Write tabs on Etherscan, letting users interact with contracts directly through the explorer.

Verification with Hardhat

Hardhat's verify task submits source code to Etherscan.

```
npx hardhat verify --network mainnet CONTRACT_ADDRESS
"Constructor" "Arguments"
```

Breaking down:

npx hardhat verify - Verification command
--network mainnet - Target network
CONTRACT_ADDRESS - Deployed contract address

"Constructor" "Arguments" - Constructor arguments in order

For contracts with complex arguments create a verification file.

```
module.exports = [  
  "MyToken",  
  "MTK",  
  "10000000000000000000000000000000"  
]
```

Then verify using the arguments file.

```
npx hardhat verify --network mainnet CONTRACT_ADDRESS  
--constructor-args arguments.js
```

Breaking down:

--constructor-args arguments.js - File containing constructor arguments

Verification with Foundry

Foundry can verify during deployment or separately.

```
forge verify-contract CONTRACT_ADDRESS src/  
Token.sol:Token --chain-id 1
```

Breaking down:

forge verify-contract - Verification command
CONTRACT_ADDRESS - Deployed address
src/Token.sol:Token - Source file and contract name
--chain-id 1 - Ethereum mainnet chain ID

With constructor arguments.


```
}
```

SECTION 7: UPGRADEABILITY

Understanding Proxy Patterns

Smart contracts are immutable. Once deployed, code cannot change. Proxy patterns provide upgradeability by separating storage from logic.

A proxy contract holds storage and delegates calls to an implementation contract. To upgrade you deploy new implementation and update the proxy's target. Storage remains intact while logic changes.

Deploying Upgradeable Contracts with OpenZeppelin

OpenZeppelin provides battle-tested upgrade infrastructure.

```
npm install @openzeppelin/contracts-upgradeable  
@openzeppelin/hardhat-upgrades
```

The upgradeable contract initializes instead of using constructor.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts-upgradeable/token/ERC20/  
ERC20Upgradeable.sol";  
import "@openzeppelin/contracts-upgradeable/access/  
OwnableUpgradeable.sol";  
import "@openzeppelin/contracts-upgradeable/proxy/utils/  
Initializable.sol";  
import "@openzeppelin/contracts-upgradeable/proxy/utils/  
UUPSUpgradeable.sol";
```

```
contract TokenV1 is Initializable, ERC20Upgradeable,
OwnableUpgradeable, UUPSUpgradeable {
```

```
    function initialize(
        string memory name,
        string memory symbol,
        uint256 initialSupply
    ) public initializer {
        _ERC20_init(name, symbol);
        _Ownable_init(msg.sender);
        _UUPSUpgradeable_init();

        _mint(msg.sender, initialSupply);
    }
```

```
    function _authorizeUpgrade(address newImplementation)
internal override onlyOwner {}
}
```

Deploy using the upgrades plugin.

```
const { ethers, upgrades } = require("hardhat")
```

```
async function deployUpgradeable() {
    const TokenV1 = await
ethers.getContractFactory("TokenV1")
```

```
    const proxy = await upgrades.deployProxy(
TokenV1,
["MyToken", "MTK", ethers.parseEther("1000000")],
{ kind: "uups" }
    )
```

```
    await proxy.waitForDeployment()
```

```
    console.log("Proxy deployed to:", await proxy.getAddress())
```

```

    const implementationAddress = await
    upgrades.erc1967.getImplementationAddress(
      await proxy.getAddress()
    )
    console.log("Implementation deployed to:",
      implementationAddress)

    return proxy
  }

```

Upgrading Contracts

Create V2 with new functionality.

```

contract TokenV2 is TokenV1 {
  uint256 public newFeature;

  function setNewFeature(uint256 value) external onlyOwner {
    newFeature = value;
  }

  function version() external pure returns (string memory) {
    return "2.0.0";
  }
}

```

Upgrade the proxy to point to V2.

```

async function upgradeToV2(proxyAddress) {
  const TokenV2 = await
  ethers.getContractFactory("TokenV2")

  const upgraded = await
  upgrades.upgradeProxy(proxyAddress, TokenV2)

  console.log("Upgraded to V2")
}

```

```

const version = await upgraded.version()
console.log("Version:", version)

return upgraded
}

```

Upgrade Safety Checks

The OpenZeppelin plugin validates upgrades are safe.

```

async function validateUpgrade(proxyAddress) {
  const TokenV2 = await
ethers.getContractFactory("TokenV2")

  await upgrades.validateUpgrade(proxyAddress, TokenV2)

  console.log("Upgrade is safe")
}

```

The validation checks for storage layout compatibility, initializer presence, and constructor absence. Incompatible changes cause validation to fail.

Common issues include adding new storage variables in wrong positions, removing or reordering existing variables, and changing variable types.

SECTION 8: MONITORING AND MAINTENANCE

On-Chain Monitoring

Monitor deployed contracts for unexpected behavior.

```

const { ethers } = require("ethers")

class ContractMonitor {

```

```
constructor(config) {  
  this.provider = new ethers.WebSocketProvider(config.wsUrl)  
  this.contracts = new Map()  
  this.alertCallback = config.onAlert  
}
```

```
addContract(name, address, abi, events) {  
  const contract = new ethers.Contract(address, abi,  
this.provider)
```

```
for (const eventName of events) {  
  contract.on(eventName, (...args) => {  
    const event = args[args.length - 1]
```

```
    this.handleEvent({  
      contract: name,  
      event: eventName,  
      args: args.slice(0, -1),  
      blockNumber: event.log.blockNumber,  
      transactionHash: event.log.transactionHash  
    })  
  })  
}
```

```
this.contracts.set(name, contract)  
}
```

```
handleEvent(eventData) {  
  console.log(`Event: ${eventData.contract}.  
${eventData.event}`)  
  console.log(`Block: ${eventData.blockNumber}`)  
  console.log(`Args:`, eventData.args)
```

```
if (this.shouldAlert(eventData)) {  
  this.alertCallback(eventData)  
}  
}
```

```

shouldAlert(eventData) {
  if (eventData.event === "OwnershipTransferred") {
    return true
  }

  if (eventData.event === "Transfer") {
    const amount = eventData.args[2]
    if (amount > ethers.parseEther("10000")) {
      return true
    }
  }

  return false
}

```

Balance Monitoring

Track key account balances.

```

async function monitorBalances(config) {
  const provider = new ethers.JsonRpcProvider(config.rpcUrl)

  const checkBalances = async () => {
    for (const account of config.accounts) {
      const balance = await provider.getBalance(account.address)
      const balanceEth = Number(ethers.formatEther(balance))

      console.log(`${account.name}: ${balanceEth.toFixed(4)}
      ETH`)

      if (balanceEth < account.minBalance) {
        await sendAlert({
          severity: "warning",
          message: `Low balance on ${account.name}`,

```

```

balance: balanceEth,
threshold: account.minBalance
})
}
}
}

```

```

await checkBalances()

```

```

setInterval(checkBalances, config.intervalMs || 60000)
}

```

Error Tracking

Capture and analyze failed transactions.

```

async function analyzeFailedTransaction(txHash, provider) {
  const tx = await provider.getTransaction(txHash)
  const receipt = await
provider.getTransactionReceipt(txHash)

```

```

if (receipt.status === 1) {
  return { success: true }
}

```

```

try {
  await provider.call({
    from: tx.from,
    to: tx.to,
    data: tx.data,
    value: tx.value,
    gasLimit: tx.gasLimit
  }, receipt.blockNumber)
} catch (error) {
  return {
    success: false,

```

```

error: error.message,
reason: error.reason || "Unknown",
gasUsed: receipt.gasUsed.toString(),
block: receipt.blockNumber
}
}

return {
success: false,
reason: "Transaction reverted without reason"
}
}

```

Automated Maintenance Tasks

Schedule regular maintenance operations.

```

class MaintenanceScheduler {
  constructor(config) {
    this.wallet = config.wallet
    this.tasks = config.tasks
    this.running = false
  }

  start() {
    this.running = true

    for (const task of this.tasks) {
      this.scheduleTask(task)
    }
  }

  stop() {
    this.running = false
  }
}

```



```

scheduleTask(task) {
  const run = async () => {
    if (!this.running) return

    try {
      console.log(`Running task: ${task.name}`)
      await task.execute(this.wallet)
      console.log(`Completed: ${task.name}`)
    } catch (error) {
      console.error(`Task failed: ${task.name}`, error)
    }

    if (this.running) {
      setTimeout(run, task.intervalMs)
    }
  }

  setTimeout(run, task.initialDelayMs || 0)
}

```

```

const scheduler = new MaintenanceScheduler({
  wallet: wallet,
  tasks: [
    {
      name: "Harvest Rewards",
      intervalMs: 24 * 60 * 60 * 1000,
      execute: async (wallet) => {
        const staking = new ethers.Contract(
          STAKING_ADDRESS,
          STAKING_ABI,
          wallet
        )
        await staking.harvest()
      },
    },
  ],
})

```

```

name: "Compound Yields",
intervalMs: 7 * 24 * 60 * 60 * 1000,
execute: async (wallet) => {
  const vault = new ethers.Contract(
    VAULT_ADDRESS,
    VAULT_ABI,
    wallet
  )
  await vault.compound()
}
}
]
})

```

SECTION 9: MULTI-CHAIN DEPLOYMENT

Deploying to Multiple Chains

Many applications deploy identical contracts across multiple chains.

```

async function deployMultiChain(networks) {
  const deployments = {}

  for (const networkName of networks) {
    console.log(`\nDeploying to ${networkName}...`)

    const networkConfig = hre.config.networks[networkName]
    const provider = new
ethers.JsonRpcProvider(networkConfig.url)
    const wallet = new ethers.Wallet(process.env.PRIVATE_KEY,
provider)

    const Token = await ethers.getContractFactory("Token",
wallet)
    const token = await Token.deploy(

```

```

"MyToken",
"MTK",
ethers.parseEther("1000000")
)

await token.waitForDeployment()
const address = await token.getAddress()

console.log(`Deployed to ${networkName}: ${address}`)

deployments[networkName] = {
  address: address,
  txHash: token.deploymentTransaction().hash,
  chainId: networkConfig.chainId
}
}

return deployments
}

```

CREATE2 for Deterministic Addresses

CREATE2 deploys contracts to the same address on every chain.

```

pragma solidity ^0.8.24;

contract DeterministicDeployer {
  event Deployed(address indexed contractAddress, bytes32 indexed salt);

  function deploy(bytes memory bytecode, bytes32 salt)
  external returns (address) {
    address deployed;

    assembly {

```

```

    deployed := create2(0, add(bytecode, 32), mload(bytecode),
salt)
}

require(deployed != address(0), "Deployment failed");

emit Deployed(deployed, salt);

return deployed;
}

function computeAddress(bytes memory bytecode, bytes32
salt) external view returns (address) {
    bytes32 hash = keccak256(
abi.encodePacked(
bytes1(0xff),
address(this),
salt,
keccak256(bytecode)
)
);

    return address(uint160(uint256(hash)));
}
}

```

Deploy the deployer first, then use it to deploy your contracts with the same salt on each chain.

Chain-Specific Configuration

Some contracts need chain-specific settings.

```

pragma solidity ^0.8.24;

contract MultiChainToken {

```

```
mapping(uint256 => address) public bridgeAddresses;
```

```
constructor(uint256 chainId, address bridge) {  
  bridgeAddresses[chainId] = bridge;  
}  
}
```

```
const CHAIN_CONFIG = {  
  1: {  
    bridge:  
    "0x1111111111111111111111111111111111111111111111111111111111111111",  
    name: "Ethereum"  
  },  
  137: {  
    bridge:  
    "0x2222222222222222222222222222222222222222222222222222222222222222",  
    name: "Polygon"  
  },  
  42161: {  
    bridge:  
    "0x3333333333333333333333333333333333333333333333333333333333333333",  
    name: "Arbitrum"  
  }  
}
```

```
async function deployWithChainConfig(networkName) {  
  const chainId = hre.config.networks[networkName].chainId  
  const config = CHAIN_CONFIG[chainId]
```

```
  const Token = await  
  ethers.getContractFactory("MultiChainToken")  
  const token = await Token.deploy(chainId, config.bridge)
```

```
  return token  
}
```

SECTION 10: EMERGENCY PROCEDURES

Pause Functionality

Include pause mechanisms for emergencies.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/utils/Pausable.sol";  
import "@openzeppelin/contracts/access/Ownable.sol";
```

```
contract PausableToken is ERC20, Pausable, Ownable {
```

```
    constructor() ERC20("Token", "TKN") Ownable(msg.sender)  
    {}
```

```
    function pause() external onlyOwner {  
        _pause();  
    }
```

```
    function unpause() external onlyOwner {  
        _unpause();  
    }
```

```
    function transfer(address to, uint256 amount) public override  
    whenNotPaused returns (bool) {  
        return super.transfer(to, amount);  
    }
```

```
    function transferFrom(address from, address to, uint256  
    amount) public override whenNotPaused returns (bool) {  
        return super.transferFrom(from, to, amount);  
    }  
}
```

Emergency Response Plan

Document procedures before emergencies occur.

Emergency Response Document

Incident Detection

1. Monitoring alerts trigger on unusual activity
2. Community reports issue in Discord
3. Security researcher contacts team

Initial Response (First 15 minutes)

1. Verify the incident is real, not false positive
2. Assemble response team
3. Pause affected contracts if possible
4. Post acknowledgment to community

Investigation (Next 1-4 hours)

1. Identify root cause
2. Assess damage scope
3. Determine if ongoing or completed
4. Document all findings

Remediation

1. Deploy fix if applicable
2. Coordinate with protocols if cross-protocol
3. Consider white-hat recovery options
4. Plan user compensation if needed

Communication

1. Post regular updates every 30 minutes
2. Be transparent about what happened
3. Explain remediation steps
4. Outline prevention measures

Recovery Procedures

Script emergency actions in advance.

```
async function emergencyPause(contractAddress) {  
  const contract = await  
  ethers.getContractAt("PausableToken", contractAddress)
```

```
  
  const paused = await contract.paused()  
  if (paused) {  
    console.log("Contract already paused")  
    return  
  }
```

```
  
  const tx = await contract.pause({  
    maxFeePerGas: ethers.parseGwei("500"),  
    maxPriorityFeePerGas: ethers.parseGwei("50")  
  })
```

```
  
  console.log("Pause transaction:", tx.hash)  
  await tx.wait()  
  console.log("Contract paused")  
}
```

```
async function emergencyWithdraw(contractAddress,  
recipient) {  
  const contract = await ethers.getContractAt("Vault",  
contractAddress)
```

```
  
  const balance = await  
  ethers.provider.getBalance(contractAddress)  
  console.log("Contract balance:", ethers.formatEther(balance))
```

```
  
  const tx = await contract.emergencyWithdraw(recipient, {  
    maxFeePerGas: ethers.parseGwei("500"),  
    maxPriorityFeePerGas: ethers.parseGwei("50")  
  })
```

```
  
  console.log("Withdraw transaction:", tx.hash)  
  await tx.wait()
```



```
console.log("Emergency withdrawal complete")
}
```

Post-Incident Review

After resolving an incident conduct a thorough review.

Post Incident Review Template

Incident Summary

- Date and time
- Duration
- Impact (users affected, funds at risk)
- Final outcome

Timeline

- When detected
- When confirmed
- When mitigated
- When fully resolved

Root Cause Analysis

- What failed
- Why it failed
- Why testing did not catch it
- Why monitoring did not catch it sooner

Remediation

- Immediate fixes applied
- Long-term fixes planned
- Process improvements

Prevention

- How to prevent similar issues
- New tests to add
- New monitoring to implement

- Changes to deployment process

CHAPTER SUMMARY

This chapter covered the complete testing and deployment lifecycle for smart contracts.

Local development environments provide instant feedback through Hardhat and Foundry. Forking mainnet enables testing against real protocol deployments.

Testing requires comprehensive coverage including unit tests, integration tests, fuzz tests, and invariant tests. Both Hardhat with JavaScript and Foundry with Solidity provide powerful testing frameworks.

Code analysis through coverage reports and static analysis tools like Slither identifies vulnerabilities before deployment.

Testnet deployment validates contracts in realistic conditions. Verification makes contracts transparent and trustworthy.

Mainnet deployment demands extensive checklists, cost estimation, and careful execution. Verification on all relevant block explorers is essential.

Upgradeability through proxy patterns enables fixing bugs and adding features. OpenZeppelin provides battle-tested upgrade infrastructure.

Monitoring and maintenance keep deployed systems healthy. Balance monitoring, error tracking, and automated maintenance tasks ensure ongoing reliability.

Multi-chain deployment extends reach while deterministic addresses via CREATE2 simplify cross-chain interactions.

Emergency procedures including pause mechanisms and documented response plans prepare for worst-case scenarios.

With testing and deployment mastered you have completed the foundational knowledge for blockchain development. The subsequent books build on this foundation with DeFi protocols, NFT systems, infrastructure scaling, security practices, and Telegram integration.

BOOK 1 COMPLETE

You now understand how blockchains work internally from hash functions to consensus mechanisms. You can write smart contracts in both Solidity and Rust. You understand major blockchain platforms and their tradeoffs. You can implement token standards and access control patterns. You can integrate wallets and build backend services. You can test thoroughly and deploy safely.

Book 2 covers DeFi and tokenomics. You will learn to integrate with Uniswap, Aave, and other protocols. You will understand yield farming, staking, and token economic design. Each book builds on the previous, leading toward complete Telegram crypto applications that connect all these pieces together.

The foundation is set. The real building begins.